

Red Hat Enterprise MRG 2
Realtime Optimierungshandbuch

Fortgeschrittene Optimierungsverfahren für die Realtime-Komponente von Red Hat Enterprise MRG

Ausgabe 3

Lana Brindley   Alison Young   Cheryn Tan

Red Hat Enterprise MRG 2 Realtime Optimierungshandbuch

Fortgeschrittene Optimierungsverfahren für die Realtime-Komponente von Red Hat Enterprise MRG
Ausgabe 3

Lana Brindley
Red Hat Engineering Content Services

Alison Young
Red Hat Engineering Content Services

Cheryn Tan
Red Hat Engineering Content Services
cheryntan@redhat.com

Rechtlicher Hinweis

Zusammenfassung

Dieses Buch enthält fortgeschrittene Optimierungsverfahren für die MRG Realtime Komponente von Red Hat Enterprise MRG, der Plattform für verteiltes Rechnen. Installationsanleitungen finden Sie im MRG Realtime Installationshandbuch.

Inhaltsverzeichnis

Vorwort

Red Hat Enterprise MRG

Dieses Buch enthält grundlegende Informationen zur Installation und Optimierung der MRG Realtime Komponente von Red Hat Enterprise MRG. Red Hat Enterprise MRG ist eine Hochleistungsplattform für verteiltes Rechnen, die aus drei Komponenten besteht:

1. *M*essaging — plattformübergreifendes und zuverlässiges Hochleistungs-Messaging unter Verwendung des Advanced Message Queuing Protocol (AMQP) Standards.
2. *R*ealtime — Konsistent niedrige Latenz und berechenbare Reaktionszeiten für Applikationen, die Latenz im Bereich von Mikrosekunden erfordern.
3. *G*rid — Verteiltes High Throughput Computing (HTC) und High Performance Computing (HPC).

Alle drei Komponenten von Red Hat Enterprise MRG sind für den Einsatz mit der Plattform konzipiert worden, können jedoch auch separat verwendet werden.

MRG Realtime

Zahlreiche Branchen und Organisationen benötigen extrem hochleistungsfähige Rechenumgebungen sowie eine niedrige und berechenbare Latenz, insbesondere die Finanz- und Telekommunikationsbranche. Latenz, oder Reaktionszeit, ist definiert als die Zeit zwischen einem Ereignis und der Systemreaktion darauf und wird in der Regel in Mikrosekunden (µs) gemessen. Für die meisten in einer Linux-Umgebung laufenden Applikationen kann eine einfache Leistungsoptimierung die Latenz in ausreichendem Maße verbessern. Für diejenigen Branchen jedoch, in denen Latenz nicht nur gering, sondern auch berechenbar und vorhersehbar sein muss, hat Red Hat jetzt einen 'drop-in' Kernel-Ersatz entwickelt, der diesen Anforderungen gerecht wird. MRG Realtime wird als Teil von Red Hat Enterprise MRG vertrieben und bietet eine nahtlose Integration mit Red Hat Enterprise Linux 6. Mit MRG Realtime haben Kunden die Möglichkeit, Latenzzeiten in ihrer Organisation zu messen, zu konfigurieren und aufzeichnen.

Über das MRG Realtime Optimierungshandbuch

Dieses Buch ist in drei Hauptabschnitte unterteilt: Erstens allgemeine Systemoptimierung, die auf einem Red Hat Enterprise Linux 6 Kernel durchgeführt werden kann, zweitens MRG Realtime spezifische Optimierung, die auf einem MRG Realtime Kernel zusätzlich zu den standardmäßigen Red Hat Enterprise Linux 6 Optimierungen durchgeführt werden sollten, und drittens ein Abschnitt über die Entwicklung und Bereitstellung Ihrer eigenen MRG Realtime Programme.

Der MRG Realtime Kernel muss installiert sein, ehe Sie mit den Optimierungsverfahren in diesem Buch beginnen können. Falls Sie den MRG Realtime Kernel noch nicht installiert haben oder Hilfe bei Installationsproblemen brauchen, lesen Sie bitte das *MRG Realtime Installationshandbuch*.

1. Dokumentkonventionen

Dieses Handbuch verwendet mehrere Konventionen, um bestimmte Wörter und Sätze hervorzuheben und Aufmerksamkeit auf bestimmte Informationen zu lenken.

In PDF- und Papierausgaben verwendet dieses Handbuch Schriftbilder des Liberation-Fonts-Sets. Das Liberation-Fonts-Set wird auch für HTML-Ausgaben verwendet, falls es auf Ihrem System installiert ist. Falls nicht, werden alternative, aber äquivalente Schriftbilder angezeigt. Beachten Sie: Red Hat Enterprise Linux 5 und die nachfolgende Versionen beinhalten das Liberation-Fonts-Set standardmäßig.

1.1. Typografische Konventionen

Es werden vier typografische Konventionen verwendet, um die Aufmerksamkeit auf bestimmte Wörter und Sätze zu lenken. Diese Konventionen und die Umstände, unter denen sie auftreten, sind folgende:

`Nichtproportional Fett`

Dies wird verwendet, um Systemeingaben hervorzuheben, einschließlich Shell-Befehle, Dateinamen und -pfade. Es wird ebenfalls zum Hervorheben von Tasten und Tastenkombinationen verwendet. Zum Beispiel:

> Um den Inhalt der Datei **`my_next_bestselling_novel`** in Ihrem aktuellen Arbeitsverzeichnis zu sehen, geben Sie den Befehl **`cat my_next_bestselling_novel`** in den Shell-Prompt ein und drücken Sie **`Enter`**, um den Befehl auszuführen.

Das oben aufgeführte Beispiel beinhaltet einen Dateinamen, einen Shell-Befehl und eine Taste. Alle werden nichtproportional fett dargestellt und alle können, dank des Kontextes, leicht unterschieden werden.

Tastenkombinationen unterscheiden sich von einzelnen Tasten durch das Pluszeichen, das die einzelnen Teile einer Tastenkombination miteinander verbindet. Zum Beispiel:

> Drücken Sie **`Enter`**, um den Befehl auszuführen.
>
> Drücken Sie **`Strg`**+**`Alt`**+**`F2`**, um zu einem virtuellen Terminal zu wechseln.

Das erste Beispiel hebt die zu drückende Taste hervor. Das zweite Beispiel hebt eine Tastenkombination hervor: eine Gruppe von drei Tasten, die gleichzeitig gedrückt werden müssen.

Falls Quellcode diskutiert wird, werden Klassennamen, Methoden, Funktionen, Variablennamen und Rückgabewerte, die innerhalb eines Abschnitts erwähnt werden, wie oben gezeigt **`nichtproportional fett`** dargestellt. Zum Beispiel:

> Zu dateiverwandten Klassen zählen **`filesystem`** für Dateisysteme, **`file`** für Dateien und **`dir`** für Verzeichnisse. Jede Klasse hat ihren eigenen Satz an Berechtigungen.

Proportional Fett

Dies kennzeichnet Wörter oder Sätze, die auf einem System vorkommen, einschließlich Applikationsnamen, Text in Dialogfeldern, beschriftete Schaltflächen, Bezeichnungen für Auswahlkästchen und Radio-Buttons, Überschriften von Menüs und Untermenüs. Zum Beispiel:

> Wählen Sie **System → Einstellungen → Maus** in der Hauptmenüleiste aus, um die **Mauseinstellungen** zu öffnen. Wählen Sie im Reiter **Tasten** auf das Auswahlkästchen **Mit links bediente Maus** und anschließend auf **Schließen**, um die primäre Maustaste von der linken auf die rechte Seite zu ändern (d.h., um die Maus auf Linkshänder anzupassen).
>
> Um ein Sonderzeichen in eine **gedit**-Datei einzufügen, wählen Sie **Anwendungen → Zubehör → Zeichentabelle** aus der Hauptmenüleiste. Wählen Sie als Nächstes **Suchen → Suchen** aus der Menüleiste der **Zeichentabelle**, geben Sie im Feld **Suchbegriff** den Namen des Zeichens ein und klicken Sie auf **Weitersuchen**. Das gesuchte Zeichen wird daraufhin in der **Zeichentabelle** hervorgehoben. Doppelklicken Sie auf dieses hervorgehobene Zeichen, um es in das Feld **Zu kopierender Text** zu übernehmen und klicken Sie anschließend auf die Schaltfläche **Kopieren**. Gehen Sie nun zurück in Ihr Dokument und wählen Sie **Bearbeiten → Einfügen** aus der **gedit**-Menüleiste.

Der oben aufgeführte Text enthält Applikationsnamen, systemweite Menünamen und -elemente,

applikationsspezifische Menünamen sowie Schaltflächen und Text innerhalb einer grafischen Oberfläche. Alle werden proportional fett dargestellt und sind anhand des Kontextes unterscheidbar.

`Nichtproportional Fett Kursiv` oder ***Proportional Fett Kursiv***

Sowohl bei nichtproportional fett als auch bei proportional fett weist ein zusätzlicher Kursivdruck auf einen ersetzbaren oder variablen Text hin. Kursivdruck kennzeichnet Text, der nicht wörtlich eingeben wird, oder angezeigten Text, der sich abhängig von den gegebenen Umständen unterscheiden kann. Zum Beispiel:

> Um sich mit einer Remote-Maschine via SSH zu verbinden, geben Sie an einem Shell-Prompt **`ssh`** ***`username@domain.name`*** ein. Falls die Remote-Maschine **`example.com`** ist und Ihr Benutzername auf dieser Maschine John lautet, geben Sie also **`ssh john@example.com`** ein.
>
> Der Befehl **`mount -o remount`** ***`file-system`*** hängt das angegebene Dateisystem wieder ein. Um beispielsweise das **`/home`**-Dateisystem wieder einzuhängen, verwenden Sie den Befehl **`mount -o remount /home`**.
>
> Um die Version des derzeit installierten Pakets zu sehen, verwenden Sie den Befehl **`rpm -q`** ***`package`***. Die Ausgabe sieht wie folgt aus: ***`package-version-release`***.

Beachten Sie die kursiv dargestellten Begriffe oben — username, domain.name, file-system, package, version und release. Jedes Wort ist ein Platzhalter entweder für Text, den Sie für einen Befehl eingeben, oder für Text, der vom System angezeigt wird.

Neben der Standardbenutzung für die Darstellung des Titels eines Werks zeigt der Kursivdruck auch die erstmalige Verwendung eines neuen und wichtigen Begriffs an. Zum Beispiel:

> Publican ist ein *DocBook* Publishing-System.

1.2. Konventionen für Seitenansprachen

Ausgaben des Terminals und Auszüge aus dem Quellcode werden visuell vom umliegenden Text hervorgehoben durch sogenannte Seitenansprachen (auch Pull-Quotes genannt).

Eine an das Terminal gesendete Ausgabe wird in den Schrifttyp **`nichtproportional Roman`** gesetzt und wie folgt dargestellt:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Auszüge aus dem Quellcode werden ebenfalls in den Schrifttyp **`nichtproportional Roman`** gesetzt, doch wird zusätzlich noch die Syntax hervorgehoben:

```
static int kvm_vm_ioctl_deassign_device(struct kvm *kvm,
                  struct kvm_assigned_pci_dev *assigned_dev)
{
         int r = 0;
         struct kvm_assigned_dev_kernel *match;

         mutex_lock(&kvm->lock);

         match = kvm_find_assigned_dev(&kvm->arch.assigned_dev_head,
                                       assigned_dev->assigned_dev_id);
         if (!match) {
                 printk(KERN_INFO "%s: device hasn't been assigned before, "
                   "so cannot be deassigned\n", __func__);
                 r = -EINVAL;
                 goto out;
         }

         kvm_deassign_device(kvm, match);

         kvm_free_assigned_device(kvm, match);

out:
         mutex_unlock(&kvm->lock);
         return r;
}
```

1.3. Anmerkungen und Warnungen

Zu guter Letzt verwenden wir drei visuelle Stile, um die Aufmerksamkeit auf Informationen zu lenken, die andernfalls vielleicht übersehen werden könnten.



Anmerkung

Eine Anmerkung ist ein Tipp, ein abgekürztes Verfahren oder ein alternativer Ansatz für die vorliegende Aufgabe. Das Ignorieren von Anmerkungen sollte keine negativen Auswirkungen haben, aber Sie verpassen so vielleicht einen Trick, der Ihnen das Leben vereinfachen könnte.



Wichtig

Die Wichtig-Schaukästen lenken die Aufmerksamkeit auf Dinge, die sonst leicht übersehen werden können: Konfigurationsänderungen, die nur für die aktuelle Sitzung gelten oder Dienste, für die ein Neustart nötig ist, bevor eine Aktualisierung wirksam wird. Das Ignorieren von Wichtig-Schaukästen würde keinen Datenverlust verursachen, kann aber unter Umständen zu Ärgernissen und Frustration führen.



Warnung

Eine Warnung sollte nicht ignoriert werden. Das Ignorieren von Warnungen führt mit hoher Wahrscheinlichkeit zu Datenverlust.

2. Hilfe bekommen und Feedback geben

2.1. Brauchen Sie Hilfe?

Falls Sie Schwierigkeiten mit einer der in diesem Handbuch beschriebenen Prozeduren haben, besuchen Sie das Red Hat Kundenportal unter http://access.redhat.com. Via Kundenportal können Sie:

- eine Knowledgebase bestehend aus Artikeln rund um technischen Support für Red Hat Produkte durchsuchen oder zu durchstöbern.
- einen Support-Case bei Red Hat Global Support Services (GSS) einreichen.
- auf weitere Produktdokumentationen zugreifen.

Red Hat unterhält außerdem eine Vielzahl von Mailing-Listen zur Diskussion über Red Hat Software und Technologie. Eine Übersicht der öffentlich verfügbaren Listen finden Sie unter https://www.redhat.com/mailman/listinfo. Klicken Sie auf den Namen einer Liste für weitere Einzelheiten zum Abonnieren dieser Liste oder um auf deren Archiv zuzugreifen.

2.2. Wir freuen uns auf Ihr Feedback!

Wenn Sie einen Fehler in diesem Handbuch finden oder eine Idee haben, wie dieses verbessert werden könnte, freuen wir uns über Ihr Feedback! Reichen Sie einen Fehlerbericht für die Komponente **Red Hat Enterprise MRG** in Bugzilla unter http://bugzilla.redhat.com/ ein.

Vergewissern Sie sich beim Einreichen eines Fehlerberichts, dass Sie die Kennung des Handbuchs mit angeben: *Realtime_Tuning_Guide*

Falls Sie uns einen Vorschlag zur Verbesserung der Dokumentation senden möchten, sollten Sie hierzu möglichst genaue Angaben machen. Wenn Sie einen Fehler gefunden haben, geben Sie bitte die Nummer des Abschnitts und einen Ausschnitt des Textes an, damit wir diesen leicht finden können.

Kapitel 1. Bevor Sie mit der Optimierung Ihres MRG Realtime Systems beginnen

MRG Realtime wurde für die Verwendung auf gut optimierten Systemen für Applikationen mit extrem hohen Determinismus-Anforderungen konzipiert. Durch Kernel-Systemoptimierung wird der Großteil der Verbesserungen beim Determinismus bereits erreicht. Zum Beispiel verbessert sorgfältige Systemoptimierung bei vielen Arbeitslasten die Konstanz der Ergebnisse um etwa 90%. Aus diesem Grund empfehlen wir üblicherweise, dass Kunden zunächst Kapitel 2, *Allgemeine Systemoptimierung* des standardmäßigen Red Hat Enterprise Linux Systems durchführen, bevor sie den Einsatz von MRG Realtime erwägen.

Dinge, die es bei der Optimierung Ihres MRG Realtime Kernels zu beachten gilt

1. Haben Sie Geduld

   Realtime-Optimierung ist ein iterativer Prozess; fast nie werden Sie einfach ein paar Variablen optimieren können und genau wissen, dass es sich um die bestmögliche Änderung handelt. Stellen Sie sich darauf ein, ein paar Tage oder Wochen lang eine Reihe von Optimierungen auszuprobieren, um sich allmählich den besten Einstellung für Ihr System anzunähern.

   Führen Sie außerdem immer lange Testläufe durch. Die Änderung eines Optimierungsparameters mit anschließendem fünfminütigen Testlauf ist keine ausreichende Validierung für einen Satz an Einstellungen. Machen Sie die Länge Ihrer Testläufe variabel und lassen Sie diese mehr als nur ein paar Minuten laufen. Versuchen Sie dies auf ein paar verschiedene Einstellungen zu beschränken, die Sie zunächst nur wenige Stunden laufen lassen. Testen Sie dieselben Einstellungen anschließend mehrere Stunden oder Tage lang, um möglichst auch Extremsituationen mit maximaler Latenz oder Ressourcenerschöpfung festzustellen.

2. Seien Sie genau

   Implementieren Sie einen Messmechanismus in Ihrer Applikation, damit Sie genau messen können, wie ein bestimmter Satz von Optimierungseinstellungen die Leistung der Applikation beeinflusst. Vage Hinweise (z.B. "Die Maus bewegt sich weniger stockend") sind in der Regel falsch und variieren von Person zu Person. Führen Sie sachliche Messungen durch und speichern Sie die Ergebnisse für die spätere Analyse.

3. Gehen Sie methodisch vor

   Die Versuchung ist groß, zwischen den Testläufen gleich mehrere Änderungen an Optimierungsvariablen vorzunehmen, aber das bedeutet auch, dass Sie keine Möglichkeit haben festzustellen, welche dieser Einstellungen Ihre Testergebnisse beeinflusst hat. Halten Sie Änderungen zwischen Testläufen am besten so gering wie möglich.

4. Seien Sie konservativ

   Auch ist die Versuchung groß, bei der Optimierung große Änderungen durchzuführen, aber es ist fast immer besser, die Änderungen nur schrittweise durchzuführen. Langfristig erweist sich die schrittweise Erhöhung von niedrigen Prioritätswerten hin zu hohen Prioritätswerten als die Vorgehensweise mit den besten Ergebnissen.

5. Seien Sie clever

   Verwenden Sie die verfügbaren Tools. Das grafische Tuna-Tool macht es einfach, Prozessoraffinitäten für Threads und Interrupts einzustellen, Thread-Prioritäten zu ändern, und Prozessoren für Applikationen zu isolieren. Mit den Befehlszeilen-Tools **`taskset`** und **`chrt`** können Sie die meisten der Aktionen durchführen, die auch Tuna bietet. Falls Sie auf Probleme mit der Leistung stoßen, kann das **`ftrace`**-Tool im Trace-Kernel dabei helfen, die Ursachen dafür zu finden.

6. Seien Sie flexibel

   Statt Werte in Ihrer Applikation hart zu kodieren, verwenden Sie externe Tools um die Werte für Richtlinie, Priorität und Affinität zu ändern. Dies ermöglicht es Ihnen, viele verschiedene

Kombinationen auszuprobieren, und es vereinfacht Ihre Logik. Wenn Sie Einstellungen gefunden haben, die zu guten Ergebnissen führen, so können Sie diese entweder Ihrer Applikation hinzufügen oder eine Startup-Logik formulieren, die die Einstellungen beim Start der Applikation implementiert.

Wie Optimierung die Leistung verbessert

Der überwiegende Teil der Leistungsoptimierung wird durch Beeinflussung von Prozessoren (Central Processing Units oder CPUs) erreicht. Prozessoren werden beeinflusst durch:

Interrupts:

Bei Software ist ein Interrupt ein Ereignis, das eine Änderung bei der Ausführung erfordert.

Interrupts werden von einem Satz Prozessoren bedient. Durch Anpassung der Affinitätseinstellung eines Interrupts können wir bestimmen, auf welchem Prozessor der Interrupt ausgeführt wird.

Threads:

Threads ermöglichen es Programmen, zwei oder mehr Aufgaben simultan auszuführen.

Threads können, wie Interrupts auch, durch die Affinitätseinstellung beeinflusst werden, die festlegt, auf welchem Prozessor der Thread läuft.

Es ist außerdem möglich, Scheduling-Priorität und Scheduling-Richtlinien einzustellen, um Threads weiter zu steuern.

Indem Sie Interrupts und Threads zu bestimmten Prozessoren zuordnen, können Sie die Prozessoren indirekt beeinflussen. Dies gibt Ihnen eine größere Kontrolle über Scheduling und Prioritäten und infolgedessen über Latenz und Determinismus.

MRG Realtime Scheduling-Richtlinien

Linux verwendet drei wesentliche Scheduling-Richtlinien:

`SCHED_OTHER` (manchmal `SCHED_NORMAL` genannt)

Dies ist die standardmäßige Thread-Richtlinie, die eine vom Kernel dynamisch gesteuerte Priorität hat. Die Priorität wird basierend auf der Thread-Aktivität angepasst. Threads mit dieser Richtlinie haben eine Echtzeitpriorität von 0 (Null).

`SCHED_FIFO` (First in, first out)

Eine Echtzeitrichtlinie mit einem Prioritätsbereich von 1 - 99, wobei 1 der niedrigste und 99 der höchste Wert ist. **`SCHED_FIFO`**-Threads besitzen immer eine höhere Priorität als **`SCHED_OTHER`**-Threads (zum Beispiel besitzt ein **`SCHED_FIFO`**-Thread mit einer Priorität von ***1*** eine höhere Priorität als *jeder* **`SCHED_OTHER`**-Thread). Jeder als **`SCHED_OTHER`**-Thread erstellte Thread besitzt eine feste Priorität und läuft, bis er entweder geblockt wird oder ein Thread mit höherer Priorität ihn unterbricht.

`SCHED_RR` (Round-Robin)

`SCHED_RR` ist eine Optimierung von **`SCHED_FIFO`**. Bei der **`SCHED_RR`**-Richtlinie haben Threads mit gleicher Priorität ein Quantum und werden nach Round-Robin-Verfahren (also reihum) eingeplant. Diese Richtlinie wird selten benutzt.

Kapitel 2. Allgemeine Systemoptimierung

Dieser Abschnitt enthält allgemeine Optimierungsverfahren, die auf einer standardmäßigen Red Hat Enterprise Linux Installation angewendet werden können. Es ist wichtig, dass diese zuerst durchgeführt werden, damit aus dem MRG Realtime Kernel Vorteile gezogen werden können.

Wir empfehlen, dass Sie die folgenden Abschnitte zuerst lesen. Sie enthalten Hintergrundinformationen dazu, wie Optimierungsparameter bearbeitet werden können, und helfen Ihnen dabei, die anderen Aufgaben in diesem Buch durchzuführen:

- Abschnitt 2.1, „Verwendung des Tuna-Interface“
- Abschnitt 2.2, „Einstellung persistenter Optimierungsparameter“

Wenn Sie bereit sind, die Optimierung zu starten, führen Sie die folgenden Schritte zuerst durch, da sie den größten Nutzen bringen:

- Abschnitt 2.3, „Einstellung von BIOS-Parametern“
- Abschnitt 2.4, „Interrupt- und Prozessbindung“
- Abschnitt 2.5, „Tipps für Dateisystemdeterminismus“

Wenn Sie anschließend bereit sind, mit der feineren Optimierung auf Ihrem System zu beginnen, fahren Sie mit den anderen Abschnitten in diesem Kapitel fort:

- Abschnitt 2.6, „Verwenden von Hardware-Uhren für System-Timestamps“
- Abschnitt 2.7, „Deaktivieren unnötiger Komponenten“
- Abschnitt 2.8, „Tipps für Swappping und Out-Of-Memory“
- Abschnitt 2.9, „Tipps für Netzwerkdeterminismus“
- Abschnitt 2.10, „Tipps zur **`syslog`**-Optimierung“
- Abschnitt 2.11, „Der PC-Card-Daemon“
- Abschnitt 2.12, „Verringerung von TCP-Leistungsspitzen“
- Abschnitt 2.13, „Reduktion des TCP-verzögerten ack-Timeouts“

Wenn Sie mit allen Optimierungsvorschlägen in diesem Kapitel fertig sind, fahren Sie mit Kapitel 3, *Realtime-spezifische Optimierung* fort.

2.1. Verwendung des Tuna-Interface

In diesem Buch werden Anleitungen zur direkten Optimierung des MRG Realtime Kernels geliefert. Das Tuna-Interface ist ein Tool, das bei diesen Änderungen hilft. Es besitzt eine grafische Oberfläche oder kann alternativ über die Befehls-Shell ausgeführt werden.

Tuna kann zur Änderung von Thread-Attributen (Scheduling-Richtlinie, Scheduler-Priorität und Prozessoraffinität) und Interrupts (Prozessoraffinität) verwendet werden. Das Tool wurde für die Verwendung auf einem laufenden System konzipiert und Änderungen werden sofort wirksam. Auf diese Weise kann jedes applikationsspezifische Mess-Tool die Systemleistung sofort nach den vorgenommenen Änderungen sehen und analysieren.



Anmerkung

Für Anleitungen zur Installation und Verwendung von Tuna werfen Sie bitte einen Blick auf das *Tuna Benutzerhandbuch*.

2.2. Einstellung persistenter Optimierungsparameter

Dieses Buch enthält zahlreiche Beispiele zu Kernel-Optimierungsparametern. Sofern nicht anders angegeben, bleiben mit den beschriebenen Verfahren die Parameter wirksam, bis das System neu gestartet wird oder die Parameter explizit geändert werden. Diese Vorgehensweise ist effektiv zur Ermittlung der anfänglichen Optimierungskonfiguration.

Nachdem Sie entschieden haben, welche Optimierungskonfiguration am besten für Ihr System funktioniert, möchten Sie wahrscheinlich, dass diese Parameter über Neustarts hinweg erhalten bleiben. Welche Methode Sie dazu wählen, hängt vom Parameter ab, den Sie einstellen.

Bearbeiten der `/etc/sysctl.conf`-Datei

Jeder mit **`/proc/sys/`** beginnende Parameter wird persistent, sobald er in die **`/etc/sysctl.conf`**-Datei eingefügt wird.

1. Öffnen Sie die **`/etc/sysctl.conf`**-Datei in Ihrem bevorzugten Texteditor.
2. Entfernen Sie das **`/proc/sys/`**-Präfix aus dem Befehl und ersetzen Sie das mittlere **`/`**-Zeichen durch ein **`.`**-Zeichen.

   Zum Beispiel: Der Befehl **`echo 2 > /proc/sys/kernel/vsyscall64`** wird so zu **`kernel.vsyscall64`**.
3. Fügen Sie den neuen Eintrag mit dem erforderlichen Parameter in die **`/etc/sysctl.conf`**-Datei ein.

```
# Enable gettimeofday(2)
kernel.vsyscall64 = 2
```

4. Führen Sie **`# sysctl -p`** aus, um die neue Konfiguration zu laden.

```
# sysctl -p
...[output truncated]...
kernel.vsyscall64 = 2
```

Bearbeiten von Dateien im `/etc/sysconfig/`-Verzeichnis

Dateien im **`/etc/sysconfig/`**-Verzeichnis können für die meisten anderen Parameter hinzugefügt werden. Da die Dateien in diesem Verzeichnis sehr unterschiedlich sein können, werden sie explizit erläutert, wo dies nötig ist.

Prüfen Sie alternativ das *Red Hat Enterprise Linux Bereitstellungshandbuch*, verfügbar auf der Red Hat Dokumentations-Website, auf Information zum **`/etc/sysconfig/`**-Verzeichnis.

Bearbeiten der `/etc/rc.d/rc.local`-Datei

Verwenden Sie diese Option nur, wenn es gar nicht anders geht!

1. Passen Sie den Befehl gemäß der Anleitung im Abschnitt Bearbeiten der /etc/sysctl.conf-Datei an.
2. Fügen Sie den neuen Eintrag mit dem erforderlichen Parameter in die **`/etc/rc.d/rc.local`**-Datei ein.

2.3. Einstellung von BIOS-Parametern

Da jedes System und jeder BIOS-Anbieter unterschiedliche Begriffe und Navigationsmethoden verwendet, enthält dieser Abschnitt nur allgemeine Informationen zu BIOS-Einstellungen. Falls Sie Schwierigkeiten haben, die erwähnte Einstellung zu finden, setzen Sie sich bitte mit dem BIOS-Anbieter in Verbindung.

Energieverwaltung

Alles, was Strom spart, indem entweder die Frequenz der Systemuhr geändert oder die CPU in verschiedene Ruhezustände gesetzt wird, kann Einfluss darauf haben, wie schnell das System auf externe Ereignisse reagiert.

Für die besten Reaktionszeiten deaktivieren Sie die Energiesparoptionen im BIOS.

Error Detection und Correction (EDAC) Einheiten

EDAC-Einheiten sind sind Geräte, die vom Error Correcting Code (ECC) Speicher gemeldete Fehler aufspüren und korrigieren. In der Regel reichen die EDAC-Optionen von keiner ECC-Prüfung bis zu einer regelmäßigen Fehlerprüfung aller Speicherknoten. Je höher die EDAC-Ebene, desto mehr Zeit wird im BIOS verbracht und desto wahrscheinlicher ist es, dass kritische Ereignis-Deadlines verpasst werden.

Falls möglich, schalten Sie EDAC ab. Falls nicht, wechseln Sie in die niedrigste, funktionale Ebene.

System Management Interrupts (SMI)

SMIs sind eine Einrichtung, mittels derer Hardware-Anbieter sicherstellen, dass das System ordnungsgemäß läuft. Der SMI-Interrupt wird in der Regel nicht vom laufenden Betriebssystem gehandhabt, sondern von Code im BIOS. SMIs werden normalerweise für Temperaturverwaltung, Remote-Konsolenverwaltung (IPMI), EDAC-Prüfungen und verschiedene andere Aufgaben zur Hardwareverwaltung verwendet.

Falls das BIOS SMI-Optionen enthält, konsultieren Sie den Anbieter und relevante Dokumentation, inwieweit es sicher ist, diese zu deaktivieren.



Warnung

Es ist zwar möglich, SMIs vollständig zu deaktivieren, allerdings wird davon dringend abgeraten. Falls Sie Ihrem System die Möglichkeit entziehen, SMIs zu generieren und zu bedienen, kann dies zu katastrophalen Hardware-Problemen führen.

2.4. Interrupt- und Prozessbindung

Echtzeitumgebungen müssen die Latenz minimieren oder eliminieren, wenn sie auf verschiedene Ereignisse reagieren. Idealerweise können Interrupts (IRQs) und Benutzerprozesse auf verschiedenen, dedizierten CPUs voneinander isoliert werden.

Interrupts werden in der Regel gleichmäßig zwischen CPUs aufgeteilt. Dadurch kann es zu verzögerter Interrupt-Bearbeitung kommen, da neue Daten und Anweisungscaches geschrieben werden müssen und es oft zu Konflikten mit anderen, in der CPU laufenden Prozessen kommt. Um dieses Problem zu beheben, können zeitkritische Interrupts und Prozesse einer bestimmten CPU (oder einer Reihe von CPUs) zugewiesen werden. Auf diese Weise ist die Wahrscheinlichkeit am höchsten, dass der Code

und die Datenstrukturen, die zur Verarbeitung dieses Interrupts notwendig sind, sich in den Prozessordaten und Anweisungscaches befinden. Der zugewiesene Prozess kann dann so schnell wie möglich laufen, während alle anderen, nicht-zeitkritischen Prozesse auf den anderen CPUs laufen. Dies kann besonders in Fällen wichtig sein, in denen die betreffenden Geschwindigkeiten in Grenzen von Speicher und verfügbarer peripherer Busbandbreite liegen. In solchen Fällen kann es zu einer merklichen Beeinträchtigung der allgemeinen Verarbeitungszeit und des Determinismus kommen, wenn es zu Wartezeiten beim Abruf von Speicher in Prozessorcaches kommt.

In der Praxis ist optimale Leistung vollständig applikationsspezifisch. Bei der Optimierung von Applikationen für verschiedene Unternehmen mit vergleichbaren Funktionen etwa erwiesen sich die jeweils optimalen Optimierungsparameter als völlig unterschiedlich. Für ein Unternehmen war die Isolation von 2 von 4 CPUs für Betriebssystemfunktionen und Interrupt-Handhabung sowie die Zuweisung der übrigen 2 CPUs für reine Applikationshandhabung optimal. Für das andere Unternehmen brachte die Bindung der netzwerkbezogenen Applikationsprozesse an eine CPU, die den Netzwerkgerätetreiber-Interrupt handhabte, einen optimalen Determinismus. Letzten Endes wird die optimale Konfiguration oft erst durch das Ausprobieren verschiedener Einstellungen erreicht, um zu sehen, was für die jeweilige Organisation am besten funktioniert.



Wichtig

Für viele der hier beschriebenen Verfahren werden Sie die CPU-Maske für eine bestimmte CPU oder Reihe von CPUs kennen müssen. Die CPU-Maske wird in der Regel als eine 32-Bit Bit-Maske (bei 32-Bit Rechnern) dargestellt. Sie kann auch als Dezimal- oder Hexadezimalzahl dargestellt werden, je nach verwendetem Befehl. Zum Beispiel: Die CPU-Maske für nur die CPU 0 ist **00000000000000000000000000000001** als Bit-Maske, **1** als Dezimalwert und **0x00000001** als Hexadezimalwert. Die CPU-Maske für CPU 0 und 1 zusammen ist **00000000000000000000000000000011** als Bit-Maske, **3** als Dezimalwert und **0x00000003** als Hexadezimalwert.

Deaktivierung des irqbalance-Daemons

Dieser Daemon ist standardmäßig aktiviert und erzwingt periodisch die Handhabung von Interrupts durch CPUs auf gleichmäßige, gerechte Weise. Allerdings sind bei Echtzeit-Deployments die Applikationen in der Regel bestimmten CPUs zugewiesen und an diese gebunden, so dass der **irqbalance**-Daemon nicht benötigt wird.

1. Prüfen Sie den Status des **irqbalance**-Daemons.

```
# service irqbalance status
irqbalance (pid PID) is running...
```

2. Falls der **irqbalance**-Daemon läuft, stoppen Sie ihn mit dem **service**-Befehl.

```
# service irqbalance stop
Stopping irqbalance:                [  OK  ]
```

3. Verwenden Sie **chkconfig**, um sicherzustellen, dass **irqbalance** beim Boot-Vorgang nicht neu startet.

```
# chkconfig irqbalance off
```

Teilweise Deaktivierung des irqbalance-Daemons

Alternativ ist es auch möglich, **irqbalance** nur auf denjenigen CPUs zu deaktivieren, die zugewiesene Funktionen besitzen, und es auf allen anderen CPUs zu aktivieren. Sie erreichen dies durch Bearbeiten der **/etc/sysconfig/irqbalance**-Datei.

1. Öffnen Sie **/etc/sysconfig/irqbalance** in Ihrem bevorzugten Texteditor und suchen Sie den Abschnitt mit dem Titel **FOLLOW_ISOLCPUS**.

```
...[output truncated]...
# FOLLOW_ISOLCPUS
#       Boolean value.  When set to yes, any setting of IRQ_AFFINITY_MASK
above
#       is overridden, and instead computed to be the same mask that is
defined
#       by the isolcpu kernel command line option.
#
#FOLLOW_ISOLCPUS=no
```

2. Aktivieren Sie FOLLOW_ISOLCPUS, indem Sie das #-Symbol vom Zeilenanfang entfernen und den Wert auf ***yes*** ändern.

```
...[output truncated]...
# FOLLOW_ISOLCPUS
#       Boolean value.  When set to yes, any setting of IRQ_AFFINITY_MASK
above
#       is overridden, and instead computed to be the same mask that is
defined
#       by the isolcpu kernel command line option.
#
FOLLOW_ISOLCPUS=yes
```

3. Dies bringt **irqbalance** dazu, dass es nur auf nicht gesondert isolierten CPUs läuft. Dies hat keine Auswirkungen auf Rechnern mit lediglich zwei Prozessoren, läuft aber wirksam auf einem Dual-Core-Rechner.

Manuelle Zuweisung von CPU-Affinität zu individuellen IRQs

1. Prüfen Sie, welcher IRQ von welchem Gerät verwendet wird, indem Sie in der **/proc/interrupts**-Datei nachsehen:

```
# cat /proc/interrupts
```

   Diese Datei enthält eine Liste von IRQs. Jede Zeile zeigt die IRQ-Nummer, die Anzahl von Interrupts, die aur jeder CPU stattgefunden hat, gefolgt vom IRQ-Typ und einer Beschreibung:

```
CPU0              CPU1
0:   26575949         11         IO-APIC-edge  timer
1:         14          7         IO-APIC-edge  i8042
...[output truncated]...
```

2. Um einen IRQ anzuweisen, nur auf einem bestimmten Prozessor zu laufen, nutzen Sie den **echo**-Befehl, um die CPU-Maske (als Hexadezimalwert) an **/proc/interrupts** zu übergeben. In diesem Beispiel weisen wir den Interrupt mit IRQ-Nummer 142 an, nur auf CPU 0 zu laufen:

```
# echo 1 > /proc/irq/142/smp_affinity
```

3. Diese Änderung wird erst wirksam, nachdem ein Interrupt stattgefunden hat. Um die Einstellungen

zu testen, generieren Sie etwas Festplattenaktivität und prüfen Sie dann die **/proc/interrupts**-Datei auf Änderungen. Gehen wir davon aus, dass ein Interrupt stattgefunden hat, sollten Sie sehen, dass Interrupts auf der gewählten CPU gestiegen sind, während die Anzahl auf den anderen CPUs unverändert ist.

Bindung von Prozessen an CPUs mittels taskset-Hilfsprogramm

Das **taskset**-Hilfsprogramm verwendet die Prozess-ID (PID) einer Aufgabe, um die Affinität einzusehen oder einzustellen oder um einen Befehl mit einer gewünschten CPU-Affinität zu starten. Um die Affinität einzustellen, erfordert **taskset** die CPU-Maske als Dezimal- oder Hexadezimalwert ausgedrückt.

1. Um die Affinität eines Prozesses einzustellen, der aktuell nicht läuft, verwenden Sie **taskset** und geben die CPU-Maske und den Prozess an. In diesem Beispiel wird **my_embedded_process** angewiesen, nur CPU 3 zu verwenden (mittels der Dezimalversion der CPU-Maske).

```
# taskset 8 /usr/local/bin/my_embedded_process
```

2. Es ist auch möglich, die CPU-Affinität für bereits laufende Prozesse einzustellen, indem Sie die **-p** (**--pid**) Option mit der CPU-Maske und die PID des zu ändernden Prozesses angeben. In diesem Beispiel wird der Prozess mit einer PID von 7013 angewiesen, nur auf CPU 0 zu laufen.

```
# taskset -p 1 7013
```



Anmerkung

Das **taskset**-Hilfsprogramm funktioniert nur, wenn Non-Uniform Memory Access (NUMA) im System nicht aktiviert ist. Für weitere Informationen hierzu siehe Abschnitt 3.6, „Non-Uniform Memory Access“.

Verwandte Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- chrt(1)
- taskset(1)
- nice(1)
- renice(1)
- sched_setscheduler(2) für eine Beschreibung des Linux Scheduling-Schemas.

2.5. Tipps für Dateisystemdeterminismus

Die Reihenfolge, in der Journaländerungen eingehen, entspricht unter Umständen nicht der Reihenfolge, in der sie auf die Festplatte geschrieben werden. Das Kernel I/O-System hat die Möglichkeit, die Reihenfolge der Journaländerungen zu ändern, meist um den verfügbaren Speicherplatz optimal zu nutzen. Journalaktivität kann durch diese Neuordnung von Änderungen und durch die Festschreibung von Daten und Metadaten Latenz verursachen. Oftmals bringen Journaling-Dateisysteme eine Verlangsamung des Systems mit sich.

Das Standard-Dateisystem von Linux-Distributionen einschließlich Red Hat Enterprise Linux 6 ist ein Journaling-Dateisystem namens **ext4**. Eine ältere, größtenteils kompatible Implementierung des Dateisystems namens **ext2** verwendet kein Journaling. Wenn Ihr Unternehmen nicht unbedingt auf

Journaling angewiesen ist, sollten Sie die Verwendung von **ext2** in Erwägung ziehen. In vielen unserer besten Benchmark-Ergebnisse nutzen wir das **ext2**-Dateisystem und geben es als eine der ersten Optimierungsempfehlungen.

Journaling-Dateisysteme wie **ext4** zeichnen die Zeit auf, zu der das letzte Mal auf eine Datei zugegriffen wurde (**atime**). Falls **ext2** für Ihr System keine geeignete Lösung ist, erwägen Sie stattdessen die Deaktivierung von **atime** unter **ext4**. Die Deaktivierung von **atime** erhöht die Leistung und senkt den Energieverbrauch, indem die Anzahl der Schreibvorgänge auf das Dateisystem-Journal verringert wird.

Deaktivierung von atime

1. Öffnen Sie die **/etc/fstab**-Datei in Ihrem bevorzugten Texteditor und suchen Sie den Eintrag für den Root-Einhängepunkt.

```
LABEL=/        /        ext4     defaults        1 1
...[output truncated]...
```

2. Fügen Sie im Optionsabschnitt die Begriffe **noatime** und **nodiratime** ein. **noatime** verhindert, dass bei jedem Lesen einer Datei dessen Zugriffs-Timestamp aktualisiert wird, und **nodiratime** verhindert, dass die Zugriffszeiten von Verzeichnis-Inodes aktualisiert werden.

```
LABEL=/        /        ext3     noatime,nodiratime        1 1
```

3. Die **tmpwatch**-Datei bei Red Hat Enterprise Linux ist standardmäßig so eingestellt, dass Dateien in **/tmp** basierend auf deren **atime**-Wert bereinigt werden. Ist dies bei Ihrem System der Fall, so führen die Anweisungen oben dazu, dass die **/tmp/***-Dateien von Benutzern täglich gelöscht werden. Dies kann behoben werden, indem **tmpwatch** mit der **--mtime**-Option gestartet wird.

```
--- /etc/cron.daily/tmpwatch.orig +++ /etc/cron.daily/tmpwatch @@ -3,6 +3,6
@@
/usr/sbin/tmpwatch 720 /var/tmp
for d in /var/{cache/man,catman}/{cat?,X11R6/cat?,local/cat?}; do
 if [ -d "$d" ]; then
- /usr/sbin/tmpwatch -f 720 "$d" + /usr/sbin/tmpwatch --mtime -f 720 "$d"
 fi
```

Verwandte Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- mkfs.ext2(8)
- mkfs.ext4(8)
- mount(8) - für Informationen zu **atime**, **nodiratime** und **noatime**
- chattr(1)

2.6. Verwenden von Hardware-Uhren für System-Timestamps

Multiprozessor-Systeme wie NUMA oder SMP haben mehrere Instanzen von Hardware-Uhren. Beim Systemstart entdeckt der Kernel die verfügbaren Taktgeber und wählt davon einen zur Verwendung aus. In der **/sys/devices/system/clocksource/clocksource0/available_clocksource**-Datei sehen Sie eine Liste aller verfügbaren Taktgeber auf Ihrem System:

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

In dem Beispiel oben sind die TSC, HPET und ACPI_PM Taktgeber verfügbar.

Den derzeit verwendeten Taktgeber finden Sie in der **`/sys/devices/system/clocksource/clocksource0/current_clocksource`**-Datei:

```
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

Wechseln der Uhr

Manchmal wird die Uhr, die für die Hauptapplikation eines Systems die beste Leistung erzielen würde, aufgrund bekannter Probleme mit dieser Uhr nicht verwendet. Nachdem alle problematischen Uhren ausgeschlossen wurden, bleibt einem System unter Umständen nur noch eine Hardware-Uhr, die den Mindestanforderungen eines Echtzeitsystems nicht gerecht werden kann.

Die Anforderungen für wichtige Applikationen unterscheiden sich von System zu System. Daher ist auch die beste Uhr für jede Applikation und somit für jedes System eine andere. Manche Applikationen erfordern eine hohe Auflösung der Zeitangaben, weshalb eine Uhr mit zuverlässigen Werten im Nanosekunden-Bereich geeignet ist. Andere Applikationen, die ihre Uhr sehr häufig ablesen, profitieren von einer Uhr mit geringem Aufwand zum Ablesen (kurze Zeit zwischen der Leseanfrage und dem Erhalt des Ergebnisses).

In all diesen Fällen ist es möglich, die vom Kernel ausgewählte Uhr zu ändern, vorausgesetzt, Sie verstehen die Auswirkungen dieses Schrittes und können eine Umgebung erzeugen, in denen die bekannten Nachteile der jeweiligen Hardware-Uhr nicht zum Tragen kommen. Wählen Sie zu diesem Zweck einen Taktgeber aus der Liste in der **`/sys/devices/system/clocksource/clocksource0/available_clocksource`**-Datei und schreiben Sie den Namen der Uhr in die **`/sys/devices/system/clocksource/clocksource0/current_clocksource`**-Datei. Der folgende Befehl legt beispielsweise HPET als den zu verwendenden Taktgeber fest:

```
# echo hpet > /sys/devices/system/clocksource/clocksource0/current_clocksource
```



Anmerkung

Eine kurze Beschreibung der üblichen Hardware-Uhren zum Vergleich ihrer jeweiligen Leistung finden Sie im *MRG Realtime Referenzhandbuch*.

Konfiguration zusätzlicher Boot-Parameter für die TSC-Uhr

Obwohl es keine Uhr gibt, die für alle Systeme ideal ist, so ist TSC jedoch generell der bevorzugte Taktgeber. Um die Zuverlässigkeit der TSC-Uhr zu verbessern, können Sie dem Kernel zusätzliche Boot-Parameter übergeben, zum Beispiel:

- ***`idle=poll`***: Hindert die Uhr am Eintritt in inaktiven Zustand.
- ***`processor.max_cstate=1`***: Hindert die Uhr am Eintritt in tiefe C-Zustände (Energiesparmodus), um sie synchron zu halten.

Beachten Sie jedoch, dass in beiden Fällen der Energieverbrauch steigt, da das System dauerhaft auf höchster Geschwindigkeit läuft.

Steuerung des Übergangs in Energiesparzustände

Moderne Prozessoren ermöglichen einen aktiven Übergang in höhere Energiesparzustände (C-Zustände) von niedrigeren Zuständen. Leider kann der Übergang von höheren Energiesparzuständen zurück in den aktiven Zustand mehr Zeit verbrauchen, als für eine Echtzeitapplikation ideal ist. Um diese Übergänge zu vermeiden, kann eine Applikation die "Power Management Quality of Service" (PM QoS) Schnittstelle verwenden.

Mithilfe der PM QoS Schnittstelle kann das System das Verhalten der Parameter ***idle=poll*** und ***processor.max_cstate=1*** (siehe Konfiguration zusätzlicher Boot-Parameter für die TSC-Uhr) emulieren, jedoch mit genauerer Steuerung der Energiesparzustände.

Wenn eine Applikation die **/dev/cpu_dma_latency** Datei geöffnet hält, hindert die PM QoS Schnittstelle den Prozessor am Eintritt in tiefe Ruhezustände und verhindert so unerwartete Latenzen beim Aufwachen. Wenn die Datei geschlossen wird, kehrt das System in einen Energiesparzustand zurück.

1. Öffnen Sie die **/dev/cpu_dma_latency**-Datei. Halten Sie den Dateideskriptor offen für die Dauer der Niedriglatenz-Operation.
2. Schreiben Sie eine 32-Bit-Zahl hinein. Diese Zahl steht für die maximale Antwortzeit in Mikrosekunden. Geben Sie für die schnellstmögliche Antwortzeit ***0*** an.

   Sehen Sie nachfolgend eine beispielhafte **/dev/cpu_dma_latency**-Datei:

```
static int pm_qos_fd = -1;

void start_low_latency(void)
{
 s32_t target = 0;

 if (pm_qos_fd >= 0)
  return;
 pm_qos_fd = open("/dev/cpu_dma_latency", O_RDWR);
 if (pm_qos_fd < 0) {
    fprintf(stderr, "Failed to open PM QOS file: %s",
            strerror(errno));
    exit(errno);
 }
 write(pm_qos_fd, &target, sizeof(target));
}

void stop_low_latency(void)
{
 if (pm_qos_fd >= 0)
    close(pm_qos_fd);
}
```

   Die Applikation ruft zunächst **start_low_latency()** auf, führt die erforderte Latenz-kritische Verarbeitung durch, und ruft anschließend **stop_low_latency()** auf.

Verwandte Handbuchseiten

Weitere Informationen diesbezüglich finden Sie in den folgenden Quellen:

- *Linux System Programming* von Robert Love

2.7. Deaktivieren unnötiger Komponenten

Dies ist eine gängige Methode zur Verbesserung der Leistung, wird jedoch oft übersehen. Dies sind einige der Komponenten und Dienste, die Sie gegebenenfalls deaktivieren können:

- Grafischer Desktop

  Vermeiden Sie grafische Oberflächen, wo diese nicht unbedingt nötig sind, insbesondere auf Servern. Um zu vermeiden, dass die Desktop-Software ausgeführt wird, öffnen Sie die **/etc/inittab**-Datei in Ihrem bevorzugten Texteditor und suchen Sie folgende Zeile:

  ```
  id:5:initdefault:
  ...[output truncated]...
  ```

  Diese Einstellung ändert das Runlevel, in den der Rechner automatisch bootet. Standardmäßig ist dieser Runlevel **5** - voller Mehrbenutzermodus mit grafischer Oberfläche. Ändern Sie die Ziffer in dieser Zeile auf **3**, um das standardmäßige Runlevel auf vollen Mehrbenutzermodus ohne grafische Oberfläche zu ändern.

  ```
  id:3:initdefault:
  ...[output truncated]...
  ```

- Mail Transfer Agents (MTA, wie z.B. Sendmail oder Postfix)

  Falls Sie Sendmail auf dem System, das Sie optimieren, nicht aktiv nutzen, so deaktivieren Sie es. Falls es jedoch benötigt wird, so stellen Sie sicher, dass es gut optimiert wurde oder verlegen Sie es auf einen separaten Rechner.

  Wichtig

  Sendmail wird zum Versenden systemgenerierter Nachrichten verwendet, die durch Programme wie cron ausgeführt werden. Dazu gehören auch Berichte, die durch Protokollierungsfunktionen wie logwatch generiert werden. Sie werden keine Nachrichten dieser Art empfangen können, wenn sendmail deaktiviert ist.

- Remote Procedure Calls (RPCs)
- Network File System (NFS)
- Maus-Dienste

  Falls Sie keine grafische Oberfläche wie Gnome oder KDE verwenden, dann benötigen Sie wahrscheinlich auch keine Maus. Entfernen Sie die Hardware und deinstallieren Sie **gpm**.

- Automatisierte Aufgaben

  Überprüfen Sie automatisierte **cron**- oder **at**-Jobs, die die Leistung beeinflussen könnten.

Vergessen Sie nicht, Applikationen von Drittanbietern sowie von externen Hardwareanbietern hinzugefügte Komponenten zu prüfen.

Verwandte Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- rpc(3)
- nfs(5)
- gpm(8)

2.8. Tipps für Swappping und Out-Of-Memory

Speicher-Swapping

Die Auslagerung von Speicherseiten auf die Festplatte (Swapping) kann in allen Umgebungen zu Latenz führen. Die beste Strategie, um eine geringe Latenz zu gewährleisten, ist es sicherzustellen, dass Ihre Systeme über ausreichenden Speicher verfügen, so dass keine Auslagerung notwendig ist. Wählen Sie die Größe Ihres physischen RAM stets passend für Ihre Applikation und Ihr System. Verwenden Sie **vmstat** zur Überwachung des Speicherverbrauchs und behalten Sie die Felder **si** (swap in) und **so** (swap out) im Auge. Diese sollten so oft wie möglich bei Null liegen.

Out of Memory (OOM)

Out of Memory (OOM) bezieht sich auf einen Rechenzustand, bei dem der gesamte verfügbare Speicher, einschließlich Swap-Space, zugewiesen wurde. In der Regel führt das zu einer System-Panik und das System funktioniert nicht länger wie erwartet. Es gibt einen Parameter, der das OOM-Verhalten in **/proc/sys/vm/panic_on_oom** steuert. Die Einstellung **1** führt bei einem OOM-Zustand zu einer Kernel-Panik. Die Einstellung **0** bewirkt, dass der Kernel bei einem OOM-Zustand eine Funktion namens **oom_killer** aufruft. In er Regel kann **oom_killer** fehlerhafte Prozesse beenden, so dass das System weiter ausgeführt werden kann.

1. Nutzen Sie den **echo**-Befehl, um den neuen Wert an **/proc/sys/vm/panic_on_oom** zu übergeben. Dies ist der einfachste Weg.

```
# cat /proc/sys/vm/panic_on_oom
0

#  echo 1 > /proc/sys/vm/panic_on_oom

# cat /proc/sys/vm/panic_on_oom
1
```

2. Es ist auch möglich, durch Anpassung von **oom_killer** zu priorisieren, welche Prozesse beendet werden. In **/proc/*PID*/** gibt es zwei Tools namens **oom_adj** und **oom_score**. Gültige Werte für **oom_adj** liegen im Bereich -16 bis +15. Dieser Wert wird zur Berechnung der 'Badness' des Prozesses verwendet, wobei ein Algorithmus verwendet wird, der unter anderem berücksichtigt, wie lange der Prozess schon läuft. Um den aktuellen **oom_killer**-Wert zu sehen, schauen Sie sich den **oom_score** für den Prozess an. **oom_killer** beendet zuerst Prozesse mit den höchsten Werten.

   Dieses Beispiel passt den **oom_score** eines Prozesses mit der PID 12465 an, um die Wahrscheinlichkeit zu verringern, dass **oom_killer** diesen beendet.

```
# cat /proc/12465/oom_score
79872

# echo -5 > /proc/12465/oom_adj

# cat /proc/12465/oom_score
78
```

3. Es gibt auch einen besonderen Wert von -17, der **oom_killer** für diesen Prozess deaktiviert. Im Beispiel unten gibt **oom_score** einen Wert von ***0*** an, wodurch angezeigt wird, dass dieser Prozess nicht beendet würde.

```
# cat /proc/12465/oom_score
78

# echo -17 > /proc/12465/oom_adj

# cat /proc/12465/oom_score
0
```

Verwandte Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- swapon(2)
- swapon(8)
- vmstat(8)

2.9. Tipps für Netzwerkdeterminismus

Transmission Control Protocol (TCP)

TCP kann sich stark auf die Latenz auswirken. TCP führt Latenzen ein, um Effizienz zu erreichen, Überlastung zu steuern und zuverlässige Übertragung sicherzustellen. Berücksichtigen Sie bei der Optimierung folgende Punkte:

- Benötigen Sie eine geordnete Übertragung?
- Müssen Sie Paketverlust vermeiden?

  Die mehrmalige Übertragung von Paketen kann zu Verzögerungen führen.
- Falls Sie auf TCP angewiesen sind, erwägen Sie die Deaktivierung des Nagle-Pufferalgorithmus durch Verwendung von **`TCP_NODELAY`** auf Ihrem Socket. Der Nagle-Algorithmus sammelt kleine, ausgehende Pakete zur gemeinsamen Versendung und kann sich nachteilig auf Latenz auswirken.

Netzwerkoptimierung

Es gibt zahlreiche Wege zur Optimierung des Netzwerks. Sehen Sie folgenden einige nützliche Verfahren:

Interrupt-Drosselung

Um den Netzwerkverkehr zu reduzieren, können Pakete gesammelt und in einen einzelnen Interrupt zusammengelegt werden.

In Systemen, die große Datenmengen übertragen und bei denen Bandbreitennutzung ein wichtiger Faktor ist, kann die Verwendung des Standardwerts oder eine verstärkte Drosselung die Bandbreitennutzung erhöhen und die Systemauslastung verringern. Für Systeme, die schnelle Antwortzeiten vom Netzwerk erfordern, wird empfohlen, die Interrupt-Drosselung zu verringern oder zu deaktivieren.

Verwenden Sie die **`-C`** (**`--coalesce`**) Option mit dem **`ethtool`**-Befehl zur Aktivierung.

Überlastung

Oftmals können I/O-Switches Gegendruck ausgesetzt sein, wobei Netzwerkdaten sich in Folge voller Puffer ansammeln.

Verwenden Sie die **`-A`** (**`--pause`**) Option mit dem **`ethtool`**-Befehl, um "pause"-Parameter zu

ändern und Netzwerküberlastung zu vermeiden.

Infiniband (IB)

Infiniband ist eine Art von Kommunikationsarchitektur, die oft dazu verwendet wird, um die Bandbreite zu erhöhen und hohe Servicequalität sowie Ausfallsicherung bereitzustellen. Es kann auch zur Verringerung von Latenz durch Remote Direct Memory Access (RDMA) Fähigkeiten verwendet werden.

Netzwerkprotokollstatistiken

Nutzen Sie die **`-s`** (**`--statistics`**) Option mit dem **`netstat`**-Befehl, um den Netzwerkverkehr zu überwachen.

Verwandte Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- ethtool(8)
- netstat(8)

2.10. Tipps zur `syslog`-Optimierung

`syslog` kann Protokollnachrichten von einer beliebigen Anzahl von Programmen über ein Netzwerk weiterleiten. Je weniger häufig dies stattfindet, desto größer ist wahrscheinlich die Transaktion in Schwebe. Ist die Transaktion sehr groß, so kann es zu einer I/O-Spitze kommen. Um dies zu vermeiden, halten Sie den Intervall möglichst klein.

Verwendung von syslogd für die Systemprotokollierung.

Der Daemon zur Systemprotokollierung, genannt **`syslogd`**, wird zum Sammeln von Nachrichten von einer Reihe verschiedener Programme verwendet. Er sammelt auch vom Kernel durch den Kernel-Protokollierungsdaemon **`klogd`** gemeldete Informationen. In der Regel protokolliert **`syslogd`** in eine lokale Datei, kann aber so konfiguriert werden, dass er über ein Netzwerk auf einen entfernten Protokollierungsserver schreibt.

1. Um dieses sogenannte Remote-Logging zu aktivieren, müssen Sie zunächst den Rechner konfigurieren, der die Protokolle empfangen soll. **`syslogd`** verwendet die in den **`/etc/sysconfig/syslog`**- und **`/etc/syslog.conf`**-Dateien definierten Konfigurationseinstellungen. Um **`syslogd`** anzuweisen, Protokolle von entfernten Rechnern zu empfangen, öffnen Sie **`/etc/sysconfig/syslog`** in Ihrem bevorzugten Texteditor und suchen Sie die **`SYSLOGD_OPTIONS=`**-Zeile.

```
# Options to syslogd
# -m 0 disables 'MARK' messages.
# -r enables logging from remote machines
# -x disables DNS lookups on messages received with -r
# See syslogd(8) for more details

SYSLOGD_OPTIONS="-m 0"

...[output truncated]...
```

2. Fügen Sie den ***-r***-Parameter an die Optionszeile an:

```
SYSLOGD_OPTIONS="-m 0 -r"
```

3. Nachdem die Remote-Logging-Unterstützung auf dem Remote-Logging-Server aktiviert wurde, muss jedes System, das Protokolle an ihn senden wird, konfiguriert werden, um seine syslog-Ausgabe an den Server zu schicken statt ins lokale Dateisystem zu schreiben. Bearbeiten Sie zu diesem Zweck die **/etc/syslog.conf**-Datei auf jedem Client-System. Für jede Protokollierungsregel in dieser Datei können Sie die lokale Protokolldatei durch die Adresse des Remote-Logging-Servers ersetzen.

```
# Log all kernel messages to remote logging host.
kern.*     @my.remote.logging.server
```

Das Beispiel oben führt dazu, dass das Client-System alle Kernel-Nachrichten auf dem entfernten Rechner unter ***@my.remote.logging.server*** protokolliert.

4. **syslogd** kann auch dahingehend konfiguriert werden, alle lokal generierten Systemnachrichten zu protokollieren, indem der **/etc/syslog.conf**-Datei eine Platzhalterzeile hinzugefügt wird:

```
# Log all messages to a remote logging server:
*.*     @my.remote.logging.server
```



Wichtig

Beachten Sie, dass **syslogd** keine integrierte Ratenbegrenzung auf dem generierten Netzwerkverkehr anwendet. Wir empfehlen Ihnen daher, die Remote-Protokollierung bei MRG Realtime Systemen auf diejenigen Nachrichten zu beschränken, die von Ihrer Organisation tatsächlich entfernt protokolliert werden müssen, so zum Beispiel Kernel-Warnungen, Authentifikationsanfragen und dergleichen. Andere Nachrichten sollten stattdessen lokal protokolliert werden.

Verwandte Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- syslog(3)
- syslog.conf(5)
- syslogd(8)

2.11. Der PC-Card-Daemon

Der **pcscd**-Daemon wird zur Verwaltung von Verbindungen mit PC- und SC-SmartCard Readern verwendet. Obwohl **pcscd** in der Regel eine niedrige Priorität hat, kann er oft mehr CPU-Zeit beanspruchen als andere Daemons. Diese zusätzlichen Hintergrundaufgaben können höheren Aufwand zur Unterbrechung dieser Aufgaben durch Echtzeitaufgaben bedeuten sowie andere unerwünschte Auswirkungen auf Determinismus haben.

Deaktivierung des pcscd-Daemons

1. Prüfen Sie den Status des **pcscd**-Daemons.

```
# service pcscd status
pcscd (pid PID) is running...
```

2. Falls der **pcscd**-Daemon läuft, stoppen Sie ihn mithilfe des **service**-Befehls.

```
# service pcscd stop
Stopping PC/SC smart card daemon (pcscd):              [  OK  ]
```

3. Verwenden Sie **chkconfig** um sicherzustellen, dass **pcscd** beim Systemneustart nicht ebenfalls wieder startet.

```
# chkconfig pcscd off
```

2.12. Verringerung von TCP-Leistungsspitzen

Um Leistungsspitzen im Zusammenhang mit der Generierung von Timestamps zu vermeiden, ändern Sie die Werte der TCP-Einträge mithilfe des **sysctl**-Befehls. Der Timestamp-Kernelparameter befindet sich unter **/proc/sys/net/ipv4/tcp_timestamps**.

- Aktivieren Sie mithilfe des folgenden Befehls die Timestamps:

```
$ sysctl -w net.ipv4.tcp_timestamps=1
net.ipv4.tcp_timestamps = 1
```

- Deaktivieren Sie mithilfe des folgenden Befehls die Timestamps:

```
$ sysctl -w net.ipv4.tcp_timestamps=0
net.ipv4.tcp_timestamps = 0
```

- Um den aktuellen Wert mithilfe des sysctl-Befehls anzuzeigen:

```
$ sysctl net.ipv4.tcp_timestamps
net.ipv4.tcp_timestamps = 1
```

Der Wert ***1*** zeigt an, dass Timestamps aktiv sind, der Wert ***0*** zeigt an, dass sie inaktiv sind.

2.13. Reduktion des TCP-verzögerten ack-Timeouts

Bei einigen Anwendungen, die kleine Netzwerkpakete verschicken, kann es aufgrund des TCP-verzögerten Acknowledgement-Timeouts zu Latenzen kommen. Dieser Wert beträgt standardmäßig 40 ms. Um dieses Problem zu umgehen, versuchen Sie den **tcp_delack_min**-Timeout-Wert zu verringern. Dies ändert die Mindestdauer, die gewartet wird, ehe eine systemweite Bestätigung verschickt wird.

1. Schreiben Sie den gewünschten Mindestwert in Mikrosekunden in **/proc/sys/net/ipv4/tcp_delack_min**:

```
# echo 1 > /proc/sys/net/ipv4/tcp_delack_min
```

Kapitel 3. Realtime-spezifische Optimierung

Wenn Sie die Optimierungen in *Kapitel 2, Allgemeine Systemoptimierung* abgeschlossen haben, so können Sie mit den MRG Realtime spezifischen Optimierungsverfahren beginnen. Hierfür muss der MRG Realtime Kernel installiert sein.



Wichtig

Versuchen Sie nicht, die Tools in diesem Abschnitt zu verwenden, ehe Sie *Kapitel 2, Allgemeine Systemoptimierung* abgeschlossen haben. Sie werden sonst keine Verbesserung der Leistung feststellen.

Wenn Sie bereit sind, die MRG Realtime Optimierung zu starten, führen Sie die folgenden Schritte zuerst durch, da sie den größten Nutzen bringen:

- Abschnitt 3.1, „Einstellung von Scheduler-Prioritäten“

Wenn Sie anschließend bereit sind, mit der feineren Optimierung auf Ihrem System zu beginnen, fahren Sie mit den anderen Abschnitten in diesem Kapitel fort:

- Abschnitt 3.2, „Verwendung von **kdump** und **kexec** mit dem MRG Realtime Kernel“
- Abschnitt 3.3, „TSC Timer-Synchronisation bei Opteron CPUs“
- Abschnitt 3.4, „Infiniband“
- Abschnitt 3.6, „Non-Uniform Memory Access“

Dieses Kapitel enthält auch Informationen zu Monitoring-Tools:

- Abschnitt 3.8, „Verwenden des **ftrace**-Hilfsprogramms zum Aufspüren von Latenzen“
- Abschnitt 3.9, „Latenz-Tracing mittels **trace-cmd**“
- Abschnitt 3.10, „Verwenden von **sched_nr_migrate** zur Einschränkung von **SCHED_OTHER**-Aufgabenmigration“

Wenn Sie mit allen Optimierungsvorschlägen in diesem Kapitel fertig sind, fahren Sie mit *Kapitel 4, Optimierung und Bereitstellung von Applikationen* fort.

3.1. Einstellung von Scheduler-Prioritäten

Der MRG Realtime Kernel ermöglicht eine sehr genaue Steuerung von Scheduler-Prioritäten. Auch ist es damit möglich, Programmen auf Applikationsebene eine höhere Priorität zuzuweisen als Kernel-Threads. Dies kann zwar hilfreich sein, birgt jedoch auch gewisse Risiken. Es kann passieren, dass das System hängen bleibt oder anderweitig unvorhergesehen reagiert, falls entscheidende Kernel-Prozesse an ihrer Ausführung gehindert werden. Letztendlich hängen die korrekten Einstellungen von der Art und Menge der Arbeitslast ab.

Prioritäten werden in Gruppen definiert, wobei einige Gruppen bestimmten Kernel-Funktionen gewidmet sind:

Tabelle 3.1. Zuordnung von Prioritäten

Priorität	Threads	Beschreibung
1	Kernel-Threads mit niedriger Priorität	Priorität 1 ist in der Regel für diejenigen Aufgaben reserviert, die nur knapp über **SCHED_OTHER** liegen müssen
2 - 69	Verfügbar	Der für typische Applikationsprioritäten verwendete Bereich
70 - 79	Soft IRQs	
80	NFS	RPC, Sperr- und Authentifikations-Threads für NFS
81 - 89	Hard IRQs	Zugehörige Threads zur Interrupt-Verarbeitung für jeden IRQ im System
90 - 98	Verfügbar	Zur *ausschließlichen* Verwendung durch sehr wichtige Applikations-Threads
99	Watchdogs und Migration	System-Threads, die mit höchster Priorität laufen müssen

Verwendung des `rtctl`-Befehls zur Einstellung von Prioritäten

1. Prioritäten werden als Reihe von Ebenen eingestellt, die von ***0*** (niedrigste Priorität) bis ***99*** (höchste Priorität) reichen. Das System-Startup-Skript **rtctl** initialisiert die Standard-Prioritäten der Kernel-Threads. Sie können die Prioritäten der verschiedenen Kernel-Threads einsehen, indem Sie den Status des **rtctl**-Dienstes anrufen.

```
# service rtctl status
2   TS      - [kthreadd]
3   FF     99 [migration/0]
4   FF     99 [posix_cpu_timer]
5   FF     50 [softirq-high/0]
6   FF     50 [softirq-timer/0]
7   FF     90 [softirq-net-tx/]
...[output truncated]...¶
```

   Die Ausgabe erfolgt im Format:

```
[PID] [scheduler policy] [priority] [process name]
```

   Im **scheduler policy**-Feld steht der Wert **TS** für die Richtlinie ***normal*** und **FF** für die Richtlinie ***FIFO***.

2. Das **rtctl** System-Startup-Skript benötigt die **/etc/rtgroups**-Datei.

   Um diese Datei zu ändern, öffnen Sie **/etc/rtgroups** in Ihrem bevorzugten Texteditor.

```
kthreads:*:1:*:\[.*\]
watchdog:f:99:*:\[watchdog.*\]
migration:f:99:*:\[migration\/.*\]
softirq:f:70:*:\[.*(softirq|sirq).*\]
softirq-net-tx:f:75:*:\[(softirq|sirq)-net-tx.*\]
softirq-net-rx:f:75:*:\[(softirq|sirq)-net-rx.*\]
softirq-sched:f:1:*:\[(softirq|sirq)-sched\/.*\]
rpciod:f:65:*:\[rpciod.*\]
lockd:f:65:*:\[lockd.*\]
nfsd:f:65:*:\[nfsd.*\]
hardirq:f:85:*:\[(irq|IRQ)[\-_/].*\]
```

3. Jede Zeile steht für einen Prozess. Sie können die Priorität der Prozesse ändern, indem Sie die Parameter wie gewünscht ändern. Die Einträge in dieser Datei folgen diesem Format:

```
[group name]:[scheduler policy]:[scheduler priority]:[regular expression]
```

Im **scheduler policy**-Feld werden die folgenden Werte akzeptiert:

o	Legt die Richtlinie ***other*** fest. Ist die Richtlinie auf **o** eingestellt, wird das **scheduler priority**-Feld auf ***0*** gesetzt und ignoriert.
b	Legt die Richtlinie ***batch*** fest.
f	Legt die Richtlinie ***FIFO*** fest.
*****	Ist die Richtlinie auf ***** eingestellt, so wird keine Änderung an übereinstimmenden Thread-Richtlinien vorgenommen.

Das **regular expression**-Feld stimmt mit dem Namen des zu modifizierenden Threads überein.

4. Nach Bearbeitung der Datei müssen Sie den **rtctl**-Dienst neu laden, damit die neuen Einstellungen wirksam werden:

```
# service rtctl stop

# service rtctl start
Setting kernel thread priorities: done
```

Zugehörige Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- rtctl(1)
- rtgroups(5)

3.2. Verwendung von kdump und kexec mit dem MRG Realtime Kernel

Kdump ist ein zuverlässiger Mechanismus zum Erstellen von Speicherauszügen im Falle eines Systemabsturzes, denn der Speicherauszug wird aus dem Kontext eines frisch gestarteten Kernels erstellt, nicht aus dem Kontext des abgestürzten Kernels. **Kdump** nutzt ein Verfahren namens **kexec** zum Booten in einen zweiten Kernel, sobald das System abstürzt. Dieser zweite Kernel, oft "Crash-Kernel" genannt, startet mit minimalem Speicher und erstellt einen Speicherauszug des Absturzes.

Ist **kdump** in Ihrem System aktiviert, so reserviert der standardmäßige Boot-Kernel einen kleinen RAM-Bereich und lädt den **kdump**-Kernel im reservierten Platz. Kommt es zu einer Kernel-Panic oder anderen schwerwiegenden Fehlern, so wird **kexec** verwendet, um in den **kdump**-Kernel zu booten, ohne über das BIOS zu gehen. Der **kdump**-Kernel bootet unter ausschließlicher Verwendung des reservierten RAM und schickt eine Fehlermeldung an die Konsole. Er schreibt anschließend einen Speicherauszug des Adressraums des Boot-Kernels in eine Datei zur späteren Fehlerdiagnose. Da **kexec** nicht über das BIOS läuft, bleibt der Speicher des ursprünglichen Boots erhalten und der Speicherauszug ist wesentlich detaillierter. Ist dies erfolgt, wird der Kernel neu gestartet, wodurch der Rechner zurückgesetzt wird und der Boot-Kernel wieder aktiv wird.



Wichtig

MRG Realtime nutzt den standardmäßigen Red Hat Enterprise Linux 6 Kernel als kdump-Kernel.

Zur Aktivierung von **kdump** unter Red Hat Enterprise Linux 6 sind drei Schritte erforderlich. Der erste Schritt stellt sicher, dass die erforderlichen RPM-Pakete auf dem System installiert sind. Der zweite Schritt erstellt die minimale Konfiguration und verändert die Grub-Befehlszeile mithilfe des **rt-setup-kdump**-Tools. Der dritte Schritte verwendet ein grafisches Systemkonfigurations-Tool namens **system-config-kdump**, um eine detailierte **kdump**-Konfiguration zu erstellen und zu aktivieren.

Installation der erforderlichen kdump-Pakete

1. Das **rt-setup-kdump**-Tool ist Teil des **rt-setup**-Pakets, das mithilfe von **yum** installiert werden kann:

   ```
   # yum install rt-setup
   ```

2. Vergewissern Sie sich, dass die **kexec-tools**- und **system-config-kdump**-Pakete installiert sind.

   ```
   # rpm -q kexec-tools system-config-kdump
   kexec-tools-2.0.0-209.el6_2.5.x86_64
   system-config-kdump-2.0.2.2-2.el6.noarch
   ```

Erstellen eines einfachen kdump-Kernels mit rt-setup-kdump

1. Starten Sie das **rt-setup-kdump**-Tool durch Aufruf im Shell-Prompt als Root-Benutzer. Dies legt den Red Hat Enterprise Linux 6 Kernel als kdump-Kernel fest:

   ```
   # rt-setup-kdump --grub
   ```

   Der ***--grub***-Parameter fügt die nötigen Änderungen in allen Realtime Kernel-Einträgen in der **/etc/grub.conf** hinzu.

2. Starten Sie das System neu, um den reservierten Speicherplatz zu implementieren. Sie können dann das **kdump** init-Skript aktivieren und den **kdump**-Dienst starten:

```
# chkconfig kdump on

# service kdump status
Kdump is not operational

# service kdump start
Starting kdump:                    [  OK  ]
```

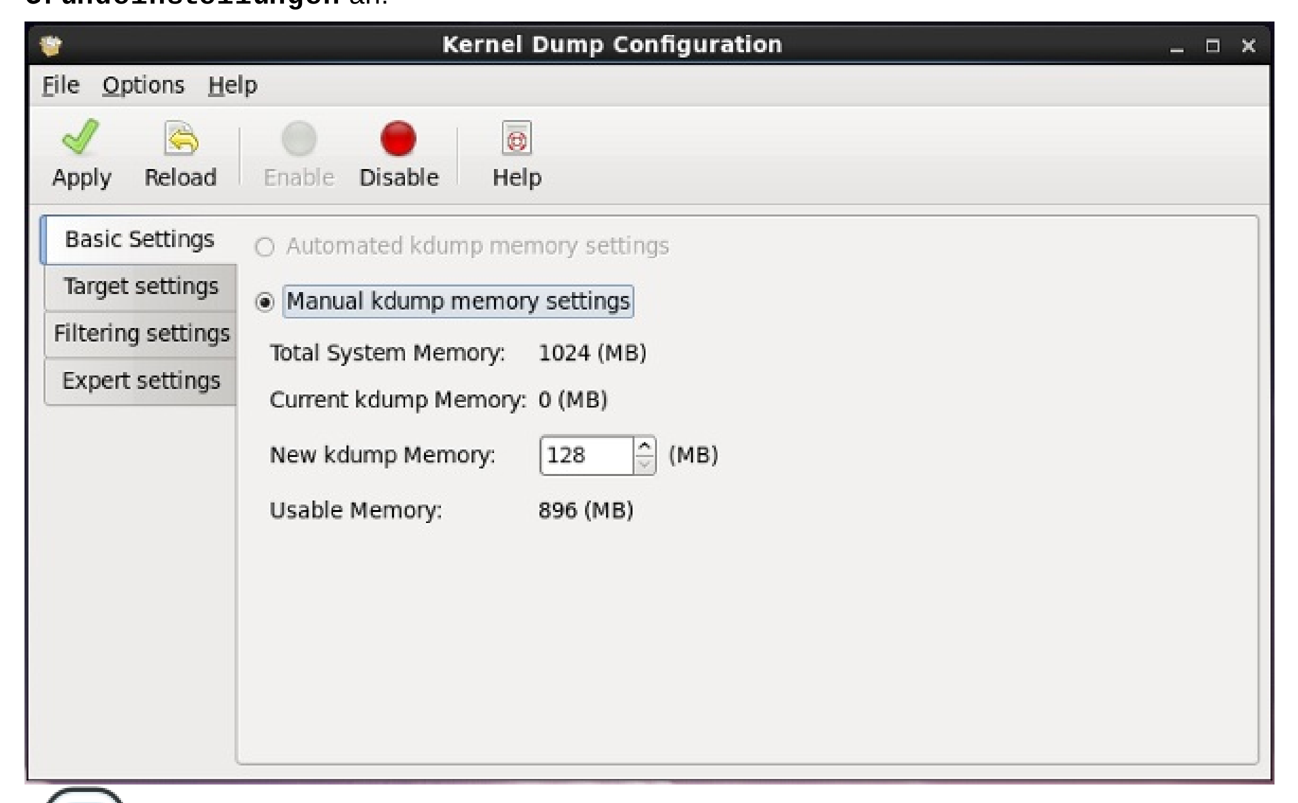
Aktivieren von kdump mit system-config-kdump

1. Wählen Sie das **Kernel Crash Dumps** Systemwerkzeug aus dem **System** → **Administration**-Menü oder verwenden Sie folgenden Befehl im Shell-Prompt:

```
# system-config-kdump
```

2. Das Fenster **Kernel-Dump-Konfiguration** wird nun angezeigt. Klicken Sie in der Werkzeugleiste auf die Schaltfläche **Aktivieren**. Der MRG Realtime Kernel unterstützt den Parameter ***crashkernel=auto***, wodurch automatisch die Speichermenge berechnet wird, die für den **kdump**-Kernel notwendig ist.

   Auf Red Hat Enterprise Linux 6 Systemen mit weniger als 4 GB RAM reserviert ***crashkernel=auto*** jedoch nicht automatisch Speicher für den **kdump**-Kernel. In diesem Fall ist es notwendig, manuell die gewünschte Speichermenge anzugeben. Geben Sie dazu den gewünschten Wert in das Feld **Neuer kdump-Speicher** auf dem Reiter **Grundeinstellungen** an:



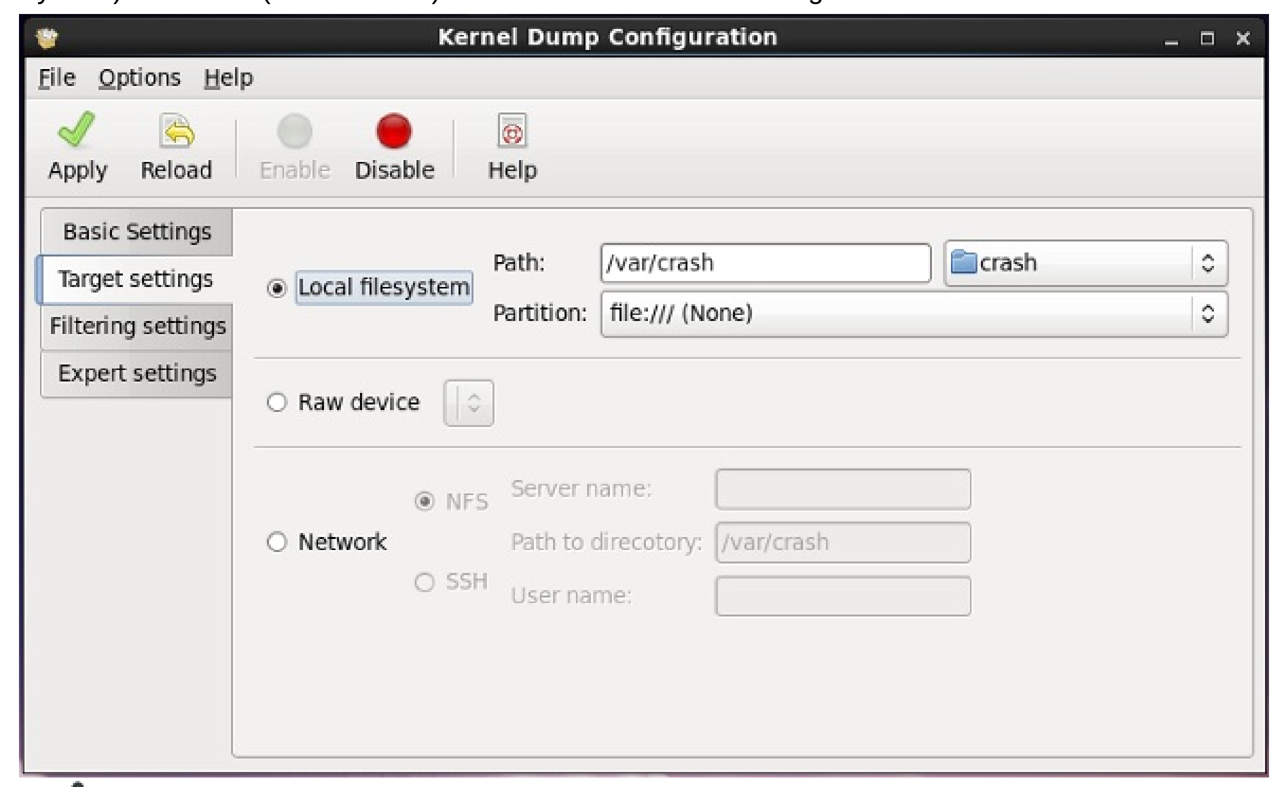


Anmerkung

Eine alternative Methode zur Zuweisung von Speicher zum **kdump**-Kernel ist das manuelle Einstellen des Parameters ***crashkernel=<value>*** in **/etc/grub.conf**.

3. Klicken Sie auf den Reiter **Ziel-Einstellungen** und spezifizieren Sie den Zielort Ihrer

Speicherauszugsdatei. Der Speicherauszug kann entweder als Datei im lokalen Dateisystem gespeichert werden, direkt an ein Gerät geschrieben werden, oder mittels NFS (Network File System) oder SSH (Secure Shell) Protokoll über ein Netzwerk gesendet werden.

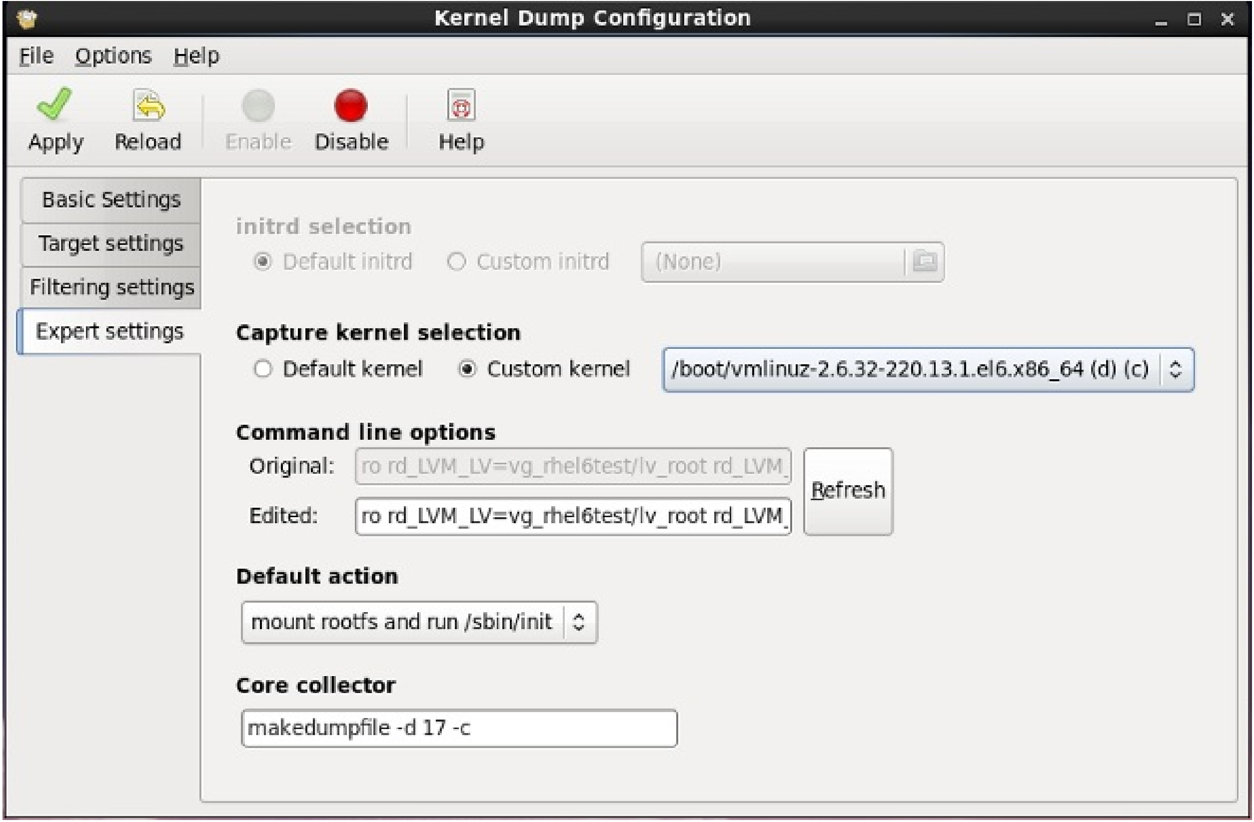




Wichtig

Überprüfen Sie stets die **/etc/grub.conf**-Datei, um sicherzustellen, dass das Tool den richtigen Kernel angepasst hat. Der MRG Realtime Kernel sollte der standardmäßige Boot-Kernel und der Red Hat Enterprise Linux Kernel sollte der Crash-Kernel sein.

4. Klicken Sie auf den Reiter **Experteneinstellungen**. Wählen Sie im Feld **Kernel-Auswahl festhalten** den **Custom-Kernel** und spezifizieren Sie den Red Hat Enterprise Linux 6 Kernel als **kdump**-Kernel.



Klicken Sie auf die Schaltfläche **Anwenden**, um Ihre Einstellungen zu speichern.

5. Starten Sie Ihr System neu, um sicherzugehen, dass **kdump** ordnungsgemäß startet. Falls Sie überprüfen möchten, ob **kdump** ordnungsgemäß funktioniert, können Sie mittels **sysrq** eine Panik simulieren:

```
# echo "c" > /proc/sysrq-trigger
```

Dies verursacht eine Kernel-Panik und das System bootet in den **kdump**-Kernel. Nachdem Ihr System mit dem Boot-Kernel wieder verfügbar ist, sollten Sie die Protokolldatei am festgelegten Speicherort prüfen können.



Anmerkung

Manche Hardware muss während der Konfiguration des **kdump**-Kernels zurückgesetzt werden. Falls Sie Probleme damit haben, den **kdump**-Kernel zum Laufen zu bringen, bearbeiten Sie die **/etc/sysconfig/kdump**-Datei und fügen Sie der **KDUMP_COMMANDLINE_APPEND**-Variable **reset_devices=1** hinzu.



Wichtig

Auf IBM LS21 Rechnern kann es beim Versuch, den kdump-Kernel zu starten, zu folgendem Warnhinweis kommen:

```
irq 9: nobody cared (try booting with the "irqpoll" option) handlers:
[<ffffffff811660a0>] (acpi_irq+0x0/0x1b)
turning off IO-APIC fast mode.
```

Einige Systeme erholen sich von diesem Fehler und setzen den Boot-Vorgang fort, während andere nach Anzeige der Nachricht hängen bleiben. Dies ist ein bekanntes Problem. Wenn Sie diesen Fehler sehen, fügen Sie die Zeile **acpi=noirq** als einen Boot-Parameter zum "kdump"-Kernel hinzu. Fügen Sie diese Zeile nur hinzu, wenn der Fehler auftritt, da sie bei Rechnern ohne dieses Problem andernfalls Boot-Probleme verursacht.

Zugehörige Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- kexec(8)
- **/etc/kdump.conf**

3.3. TSC Timer-Synchronisation bei Opteron CPUs

Die aktuelle Generation von AMD64 Opteron Prozessoren kann anfällig für einen großen **gettimeofday**-Versatz sein. Dies passiert, wenn sowohl **cpufreq** als auch der Time Stamp Counter (TSC) in Gebrauch sind. MRG Realtime bietet eine Methode dem vorzubeugen, indem alle Prozessoren simultan zum Wechsel in dieselbe Frequenz gezwungen werden. Infolgedessen erhöht sich der TSC auf einem einzelnen Prozessor nie abweichend vom TSC auf einem anderen Prozessor.

Aktivierung der TSC Timer-Synchronisation

1. Öffnen Sie die **/etc/grub.conf**-Datei in Ihrem bevorzugten Texteditor und fügen Sie dem MRG Realtime Kernel den Parameter ***clocksource=tsc powernow-k8.tscsync=1*** hinzu. Dies erzwingt die Verwendung des TSC und aktiviert simultane Kernprozessor-Frequenzwechsel.

```
...[output truncated]...
title Red Hat Enterprise Linux (realtime) (kernel-rtversion)
  root (hd0,0)
  kernel /vmlinuz-kernel-rtversion ro root=/dev/RHEL6/Root clocksource=tsc
powernow-k8.tscsync=1
  initrd /initrd-kernel-rtversion.img
```

2. Sie müssen Ihr System neu starten, damit die Änderungen wirksam werden.

Zugehörige Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- gettimeofday(2)

3.4. Infiniband

Infiniband ist eine Art von Kommunikationsarchitektur, die oft dazu verwendet wird, um die Bandbreite zu erhöhen und hohe Servicequalität sowie Ausfallsicherung bereitzustellen. Es kann auch zur Verringerung von Latenz durch Remote Direct Memory Access (RDMA) Fähigkeiten verwendet werden.

Die Unterstützung für Infiniband unter MRG Realtime unterscheidet sich nicht von der unter Red Hat Enterprise Linux 6 angebotenen Unterstützung.



Anmerkung

Weitere Informationen diesbezüglich finden Sie in Douglas Ledford's Artikel Getting Started with Infiniband, beachten Sie jedoch, dass MRG Realtime nicht Red Hat Enterprise Linux 5 unterstützt.

3.5. RoCEE und Hochleistungsnetzwerke

RoCEE (RDMA over Converged Enhanced Ethernet) ist ein Protokoll, das Remote Direct Memory Access (RDMA) über 10 Gigabit Ethernet-Netzwerke implementiert. Dies ermöglicht es Ihnen, in Ihren Rechenzentren eine konsistente Hochgeschwindigkeitsumgebung und gleichzeitig deterministische Niedriglatenz-Datenübertragung für kritische Transaktionen bereitzustellen.

High Performance Networking (HPN) umfasst eine Reihe von gemeinsam verwendeten Bibliotheken, die RoCEE-Schnittstellen zum Kernel liefern. Statt über unabhängige Netzwerkstruktur zu laufen, platziert HPN die Daten direkt in den Speicher des entfernten Systems unter Verwendung einer standardmäßigen 10 Gigabit Ethernet-Infrastruktur, wodurch niedrigerer CPU-Overhead und geringere Infrastrukturkosten erreicht werden.

Die Unterstützung für RoCEE und HPN unter MRG Realtime unterscheidet sich nicht von der unter Red Hat Enterprise Linux 6 angebotenen Unterstützung.



Anmerkung

Weitere Informationen über das Einrichten von Ethernet-Netzwerken finden Sie im Kapitel *Netzwerkschnittstellen* im *Red Hat Enterprise Linux 6 Bereitstellungshandbuch*.

3.6. Non-Uniform Memory Access

Non-Uniform Memory Access (NUMA) ist ein Verfahren zur Zuweisung von Speicherressourcen an eine bestimmte CPU. Dies kann Zugriffszeiten verbessern und führt zu selteneren Speichersperren. Obwohl dies nahelegt, dass dieses Verfahren nützlich zur Senkung von Latenzen sein kann, so hat sich gezeigt, dass NUMA-Systeme schlecht mit Echtzeitsystemen interagieren, da sie zu unerwarteten Ereignislatenzen führen können.

Wie im Abschnitt Bindung von Prozessen an CPUs mittels **`taskset`**-Hilfsprogramm erwähnt, funktioniert das **`taskset`**-Hilfsprogramm nur, wenn NUMA im System nicht aktiviert ist. Wenn Sie Prozess-Binding in Verbindung mit NUMA durchführen möchten, verwenden Sie den **`numactl`**-Befehl statt **`taskset`**.

Weitere Informationen zur NUMA-Programmierschnittstelle finden Sie in Andi Kleens Whitepaper An NUMA API for Linux.

Zugehörige Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- numactl(8)

3.7. Einhängen von debugfs

debugfs ist ein Dateisystem, das speziell für das Debugging und die Bereitstellung von Informationen für Benutzer konzipiert wurde. Hängen Sie **debugfs** für die Verwendung mit den MRG Realtime Funktionen **ftrace** und **trace-cmd** ein.

1. Hängen Sie den Kernel unter **/sys/kernel/debug** ein und weisen Sie ihn an, das **debugfs**-Dateisystem zu verwenden.

```
# mount -t debugfs nodev /sys/kernel/debug
```

2. Sie können das **debugfs**-Verzeichnis beim Boot-Vorgang automatisch einhängen, indem Sie die **/etc/fstab**-Datei in Ihrem bevorzugten Texteditor öffnen und folgende Zeile hinzufügen:

```
nodev /sys/kernel/debug    debugfs   defaults   0  0
```

3.8. Verwenden des ftrace-Hilfsprogramms zum Aufspüren von Latenzen

Eines der Diagnose-Tools, die mit dem MRG Realtime Kernel geliefert werden, ist **ftrace**, das Entwickler zur Analyse und Behebung von Latenzen und Leistungsproblemen nutzen können, die außerhalb des User-Space auftreten. Das **ftrace**-Hilfsprogramm besitzt eine Fülle von Optionen, die eine Verwendung des Tools auf unterschiedlichste Weise gestatten. Es kann dazu verwendet werden, Kontextwechsel aufzuspüren, die Aufwachzeit von höher priorisierten Aufgaben zu messen, die Zeitspanne der Deaktivierung von Interrupts zu messen oder alle während eines bestimmten Zeitraums ausgeführten Kernel-Funktionen aufzulisten.

Einige Tracer, so etwa der Funktions-Tracer, produzieren gewaltige Mengen an Daten, wodurch die Trace-Protokollanalyse sehr zeitintensiv wird. Es ist jedoch möglich, den Tracer dazu anzuweisen, nur zu arbeiten, während die Applikation kritische Code-Pfade durchläuft.

Das **ftrace**-Hilfsprogramm kann eingerichtet werden, sobald die **trace**-Variante des MRG Realtime Kernels installiert wurde und verwendet wird.

Verwendung des ftrace-Hilfsprogramms

1. Im **/sys/kernel/debug/tracing/**-Verzeichnis befindet sich eine Datei namens **available_tracers**. Diese Datei enthält alle verfügbaren Tracer für die installierte Version von **ftrace**. Um die Liste verfügbarer Tracers zu sehen, verwenden Sie den **cat**-Befehl zur Ansicht der Dateiinhalte:

```
# cat /sys/kernel/debug/tracing/available_tracers
wakeup preemptirqsoff preemptoff irqsoff ftrace sched_switch none
```

   wakeup
   Verfolgt die maximale Latenz zwischen der Reaktivierung der Prozesse mit höchster Priorität und deren Scheduling. Nur RT-Aufgaben werden von diesem Tracer berücksichtigt (**SCHED_OTHER**-Aufgaben werden ab jetzt ignoriert).

preemptirqsoff
: Verfolgt die Bereiche, die Prozessunterbrechung und Interrupts deaktivieren und speichert die maximale Zeitdauer, für die Prozessunterbrechung und Interrupts deaktiviert waren.

preemptoff
: Ähnlich dem **preemptirqsoff**-Tracer, verfolgt jedoch nur die maximale Zeitspanne, in der die Prozessunterbrechung deaktiviert war.

irqsoff
: Ähnlich wie der **preemptirqsoff**-Tracer, verfolgt jedoch nur die maximale Zeitspanne, in der Interrupts deaktiviert waren.

ftrace
: Speichert die während einer Tracing-Sitzung aufgerufenen Kernel-Funktionen. Das **ftrace**-Hilfsprogramm kann gleichzeitig mit jedem anderen Tracer laufen, ausgenommen dem **sched_switch**-Tracer.

sched_switch
: Verfolgt Kontextwechsel zwischen Aufgaben.

none
: Deaktiviert Tracing.

2. Um eine Tracing-Sitzung manuell zu starten, wählen Sie zunächst den zu verwendenden Tracer aus der Liste in **available_tracers** und verwenden Sie dann den **echo**-Befehl, um den Namen des Tracers in **/sys/kernel/debug/tracing/current_tracer** einzufügen:

```
# echo preemptoff > /sys/kernel/debug/tracing/current_tracer
```

3. Um zu prüfen, ob das **ftrace**-Hilfsprogramm aktiviert ist, verwenden Sie den **cat**-Befehl zur Ansicht der **/proc/sys/kernel/ftrace_enabled**-Datei. Ein Wert von ***1*** zeigt an, dass **ftrace** aktiviert ist und ***0*** zeigt an, dass es deaktiviert wurde.

```
# cat /proc/sys/kernel/ftrace_enabled
1
```

   Der Tracer ist standardmäßig aktiviert. Um den Tracer an- oder auszuschalten, nutzen Sie den **echo**-Befehl, um den entsprechenden Wert an die **/proc/sys/kernel/ftrace_enabled**-Datei zu übergeben.

```
# echo 0 > /proc/sys/kernel/ftrace_enabled

# echo 1 > /proc/sys/kernel/ftrace_enabled
```



Wichtig

Bei Verwendung des **echo**-Befehls sollten Sie sicherstellen, dass Sie eine Leerstelle zwischen den Wert und das **>**-Zeichen platzieren. In der Shell-Eingabeaufforderung bezieht sich die Verwendung von **0>, 1> und 2>** (ohne Leerstelle) auf Standardeingabe, Standardausgabe und Standardfehler. Deren fehlerhafte Verwendung kann unerwartete Trace-Ausgaben zur Folge haben.

4. Passen Sie Details und Parameter der Tracer an, indem Sie die Werte der verschiedenen Dateien im **/debugfs/tracing/**-Verzeichnis anpassen. Einige Beispiele sind:

   Die irqsoff, preemptoff, preempirqsoff und wakeup Tracer überwachen kontinuierlich die Latenzen. Wird eine Latenz aufgespürt, die größer als die in **tracing_max_latency** aufgezeichnete Latenz ist, so wird der Trace dieser Latenz gespeichert und **tracing_max_latency** wird entsprechend der neuen Maximalzeit aktualisiert. Auf diese Weise zeigt **tracing_max_latency** stets die seit dem Zurücksetzen höchste gemessene Latenz an.

   Um die Maximallatenz zurückzusetzen, verwenden Sie den **echo**-Befehl, um ***0*** an die **tracing_max_latency**-Datei zu übergeben. Um nur Latenzen zu sehen, die größer als der festgesetzte Wert sind, übergeben Sie mit dem **echo**-Befehl den Wert in Mikrosekunden:

   ```
   # echo 0 > /sys/kernel/debug/tracing//tracing_max_latency
   ```

   Ist der Tracing-Schwellenwert erreicht, setzt dies die Einstellung für Maximallatenz außer Kraft. Wird eine Latenz gespeichert, die größer als der Schwellenwert ist, so wird diese unabhängig von der Maximallatenz aufgezeichnet. Bei Überprüfung der Trace-Datei wird nur die zuletzt aufgezeichnete Latenz angezeigt.

   Um den Schwellenwert einzustellen, übergeben Sie mit dem **echo**-Befehl die Anzahl an Mikrosekunden, oberhalb derer Latenzen aufgezeichnet werden sollen:

   ```
   # echo 200 > /sys/kernel/debug/tracing//tracing_thresh
   ```

5. Trace-Protokolle ansehen:

   ```
   # cat /sys/kernel/debug/tracing/trace
   ```

6. Um die Trace-Protokolle zu speichern, kopieren Sie diese in eine andere Datei:

   ```
   # cat /sys/kernel/debug/tracing/trace > /tmp/lat_trace_log
   ```

7. Das **ftrace**-Hilfsprogramm kann durch Änderung der Einstellungen in der **/sys/kernel/debug/tracing/set_ftrace_filter**-Datei gefiltert werden. Sind keine Filter in der Datei festgelegt, werden alle Prozesse verfolgt. Um die aktuellen Filter einzusehen, verwenden Sie **cat**:

   ```
   # cat /sys/kernel/debug/tracing/set_ftrace_filter
   ```

8. Um die Filter zu ändern, übergeben Sie mit dem **echo**-Befehl den Namen der zu verfolgenden Funktion. Der Filter erlaubt die Verwendung eines *****-Platzhalters am Anfang oder Ende eines Suchbegriffs.

   Der *****-Platzhalter kann ebenfalls am Anfang *und* am Ende eines Worts verwendet werden. Zum Beispiel: ***irq*** wählt alle Funktionen, die **irq** im Namen enthalten.

Wird der Suchbegriff und der Platzhalter in doppelte Anführungszeichen gesetzt, verhindert dies, dass die Shell nicht versuchen wird, die Suche auf das aktuelle Arbeitsverzeichnis zu erweitern.

Sehen Sie nachfolgend einige Beispiele für Filter:

- Nur den **schedule**-Prozess verfolgen:

```
# echo schedule > /sys/kernel/debug/tracing/set_ftrace_filter
```

- Alle auf **lock** endenden Protokolle verfolgen:

```
# echo "*lock" > /sys/kernel/debug/tracing/set_ftrace_filter
```

- Alle mit **spin_** beginnenden Prozesse verfolgen:

```
# echo "spin_*" > /sys/kernel/debug/tracing/set_ftrace_filter
```

- All Prozesse mit **cpu** im Namen verfolgen:

```
# echo "*cpu*" > /sys/kernel/debug/tracing/set_ftrace_filter
```



Anmerkung

Wenn Sie ein einzelnes **>** mit dem **echo**-Befehl verwenden, so setzt dies den bestehenden Wert in der Datei außer Kraft. Wenn Sie den Wert an die Datei anhängen möchten, verwenden Sie stattdessen **>>**.

3.9. Latenz-Tracing mittels trace-cmd

trace-cmd ist eine MRG Realtime Funktion, die alle Kernel-Funktionsaufrufe sowie einige spezielle Ereignisse verfolgt. Sie zeichnet alles auf, was während eines kurzen Zeitraums im System geschieht und liefert Informationen, die zur Analyse des Systemverhaltens verwendet werden können.

Das **trace-cmd**-Tool ist in der Produktionsversion des MRG Realtime Kernels nicht aktiviert, da es zusätzlichen Overhead mit sich bringt. Falls Sie das **trace-cmd**-Tool verwenden möchten, so müssen Sie entweder die **trace** oder **debug** Variante des MRG Realtime Kernels herunterladen und installieren.



Anmerkung

Anleitungen zur Installation von Kernel-Varianten finden Sie im *MRG Realtime Installationshandbuch*.

1. Falls Sie die **trace**- oder **debug**-Varianten des MRG Realtime Kernels verwenden, so können Sie mittels **yum** das **trace-cmd**-Tool installieren.

```
# yum install trace-cmd
```

2. Um das Hilfsprogramm zu starten, geben Sie im Shell-Prompt **trace-cmd** samt der benötigten Optionen in der folgenden Syntax ein:

```
# trace-cmd [command]
```

Die Verwendung der ***-f***-Option legt das Funktions-Tracing fest und kann mit jedem anderen "trace"-Befehl verwendet werden.

Die Befehle weisen **trace-cmd** an, auf bestimmte Weise zu verfolgen.

Befehl	Beschreibung
record	Trace in einer trace.dat Datei speichern.
start	Tracing beginnen, ohne in eine Datei zu speichern.
extract	Trace vom Kernel extrahieren.
stop	Aufzeichnung von Trace-Daten durch den Kernel stoppen.
reset	Sämtliches Kernel-Tracing deaktivieren und die Trace-Puffer löschen.
report	Den in deiner trace.dat Datei gespeicherten Trace auslesen.
split	trace.dat Datei analysieren und als kleinere Datei(en) speichern.
listen	Auf einem Netzwerk-Socket auf Trace-Clients warten.
list	Verfügbare Ereignisse, Plugins oder Optionen auflisten.

Befehl	Trace-Typ	Beschreibung
-s	Kontextwechsel	Verfolgt die Kontextwechsel zwischen Tasks.
-i	Interrupts deaktiviert	Zeichnet die maximale Zeit auf, die ein Interrupt deaktiviert ist. Wird ein neues Maximum aufgezeichnet, so ersetzt es das vorherige Maximum.
-p	Prozessunterbrechung deaktiviert	Zeichnet die maximale Zeit auf, die Prozessunterbrechung deaktiviert ist. Wird ein neues Maximum aufgezeichnet, so ersetzt es das vorherige Maximum.
-b	Prozessunterbrechung und Interrupts deaktiviert	Zeichnet die maximale Zeit auf, die Prozessunterbrechung *oder* Interrupts deaktiviert sind. Wird ein neues Maximum aufgezeichnet, so ersetzt es das vorherige Maximum.
-w	Reaktivierung	Tract und speichert die maximale Zeit zum Scheduling der Aufgabe mit höchster

		Priorität nach deren Reaktivierung.
-e	Ereignis-Tracing	
-f	Funktions-Tracing	Kann mit jedem anderen Trace verwendet werden
-l	Gibt die Protokolle im **latency_trace**-Format aus	Kann mit jedem anderen Trace verwendet werden



Anmerkung

Weitere Informationen über Ereignis-Tracing und Funktions-Tracer finden Sie in *Anhang A, Ereignis-Tracing* und *Anhang B, Funktions-Tracer*.

3. In diesem Beispiel wird das **trace-cmd**-Hilfsprogramm einen einzigen Trace-Punkt untersuchen:

```
# ./trace-cmd record -e sched_wakeup ls /bin
```

3.10. Verwenden von sched_nr_migrate zur Einschränkung von SCHED_OTHER-Aufgabenmigration

Bringt eine **SCHED_OTHER**-Aufgabe eine große Anzahl anderer Aufgaben hervor, so laufen alle auf derselben CPU. Die Migrationsaufgabe oder **softirq** werden versuchen, diese Aufgaben so zu verteilen, dass sie auf inaktiven CPUs laufen. Die **sched_nr_migrate**-Option kann so eingestellt werden, dass sie die Anzahl von Aufgaben festlegt, die zu einem bestimmten Zeitpunkt verschoben werden. Da Echtzeitaufgaben auf andere Art migrieren, betrifft sie dies nicht direkt. Wenn aber **softirq** die Aufgaben verschiebt, so sperrt es den Warteschlangen-Spinlock, der zur Deaktivierung von Interrupts benötigt wird. Muss eine große Anzahl an Aufgaben verschoben werden, erfolgt dies, während Interrupts deaktiviert sind, so dass gleichzeitig keine Timer-Ereignisse oder Reaktivierungen von Prozessen erfolgen. Dies kann zu schwerwiegenden Latenzen bei Echtzeitaufgaben führen, wenn **sched_nr_migrate** auf einen hohen Wert gesetzt ist.

Anpassen der sched_nr_migrate-Variable

1. Die Erhöhung der **sched_nr_migrate**-Variable ermöglicht hohe Leistung von **SCHED_OTHER**-Threads, die viele Aufgaben hervorbringen, jedoch auf Kosten von Echtzeitlatenzen. Für niedrige Latenz von Echtzeitaufgaben auf Kosten der Leistung von **SCHED_OTHER**-Aufgaben sollte der Wert verringert werden. Der Standardwert ist 8.
2. Um den Wert der **sched_nr_migrate**-Variable anzupassen, können Sie den Wert mithilfe des **echo**-Befehls direkt an **/proc/sys/kernel/sched_nr_migrate** übergeben:

```
# echo 2 > /proc/sys/kernel/sched_nr_migrate
```

Kapitel 4. Optimierung und Bereitstellung von Applikationen

Dieses Kapitel enthält Tipps zur Verbesserung und Entwicklung von MRG Realtime Applikationen.



Anmerkung

Versuchen Sie generell, durch *POSIX* (Portable Operating System Interface) definierte Programmierschnittstellen zu verwenden. Auch die MRG Realtime Entwickler halten die POSIX-Standards ein und die Latenzreduktion im MRG Realtime Kernel basiert ebenfalls auf POSIX.

Weitere Informationsquellen

Falls Sie mehr Informationen über die Entwicklung Ihrer eigenen MRG Realtime Applikationen lesen möchten, ist dieser RTWiki Artikel ein guter Anfang.

4.1. Signalverarbeitung in Echtzeitapplikationen

Traditionelle UNIX™- und POSIX-Signale haben zwar ihre Daseinsberechtigung (insbesondere bei der Fehlerbehandlung), allerdings eignen sie sich nicht für den Einsatz in Echtzeitapplikationen als Mechanismus zur Übertragung von Ereignissen. Dies liegt daran, dass der gegenwärtige Linux-Kernelcode zur Signalhandhabung recht komplex ist, aufgrund von althergebrachtem Verhalten und der Vielzahl von Programmierschnittstellen, die unterstützt werden müssen. Diese Komplexität bedeutet, dass die Codepfade, die zur Signalübertragung durchlaufen werden müssen, nicht immer optimal sind und recht lange Latenzen für Applikationen verursachen kann.

Der ursprüngliche Grund für UNIX™-Signale war das Multiplexen eines Kontroll-Threads (der Prozess) auf verschiedene Ausführungs-Threads. Signale verhalten sich ähnlich wie Betriebssystem-Interrupts - wird ein Signal an eine Applikation übertragen, so wird der Kontext der Applikation gespeichert und sie beginnt die Ausführung eines zuvor registrierten Signal-Handlers. Ist der Signal-Handler fertig, so kehrt die Applikation zurück zu der Ausführung, bei der sie beim Empfang des Signals stehen geblieben war. In der Praxis kann dies recht kompliziert werden.

Signale sind zu nicht-deterministisch, um ihnen in einer Echtzeitapplikation vertrauen zu können. Eine bessere Option ist die Verwendung von POSIX-Threads (pthreads) zur Verteilung der Arbeitslast und zur Kommunikation zwischen verschiedenen Komponenten. Mithilfe der pthreads-Mechanismen Mutexes, Bedingungsvariablen und Barrieren können Sie Gruppen von Threads koordinieren und darauf vertrauen, dass die Codepfade durch diese relativ neuen Konstrukte wesentlich sauberer sind als beim traditionellen Code zur Handhabung von Signalen.

Weitere Informationsquellen

Weitere Informationen diesbezüglich finden Sie unter den folgenden Links:

RTWikis Build an RT Application

Ulrich Dreppers Requirements of the POSIX Signal Model

4.2. Verwendung von `sched_yield` und anderen Synchronisationsmechanismen

Der **`sched_yield`**-Systemaufruf wird von einem Thread verwendet, damit andere Threads Gelegenheit haben zu laufen. Wenn **`sched_yield`** verwendet wird, wird der Thread oft am Ende der

Ausführungswarteschleife eingereiht, so dass es eine lange Zeit dauert, bis er wieder an der Reihe ist, oder aber er wird sofort wieder zur Ausführung eingeplant, wodurch es zu einem "Busy Loop" (einer aktiven Warteschleife) auf der CPU kommt. Der Scheduler kann besser bestimmen, wann und ob andere Threads laufen wollen. Vermeiden Sie die Verwendung von **sched_yield** auf Echtzeitaufgaben.

POSIX-Threads (Pthreads) besitzen Abstraktionen, die konsistenteres Verhalten über Kernel-Versionen hinweg bieten. Das kann jedoch auch bedeuten, dass das System weniger Zeit hat, Netzwerkpakete zu verarbeiten, so dass es zu maßgeblichen Leistungseinbußen kommt. Die Art von Einbußen können schwierig zu diagnostizieren sein, da es keine signifikanten Änderungen in den Netzwerkkomponenten des Systems gibt. Bei einigen Applikationen kann es auch zu verändertem Verhalten kommen.

Weitere Informationen finden Sie in Arnaldo Carvalho de Melos Abhandlung Earthquaky kernel interfaces.

Zugehörige Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- pthread.h(P)
- sched_yield(2)
- sched_yield(3p)

4.3. Mutex-Optionen

Standardmäßige Mutex-Erstellung

Algorithmen zum wechselseitigen Ausschluss (engl. "mutual exclusion", daher kurz "Mutex"-Algorithmen) werden verwendet, um Prozesse an der simultanen Verwendung einer gemeinsamen Ressource zu hindern. Eine schnelle User-Space Mutex (engl. "fast user-space mutex", daher kurz "futex") ist ein Verfahren, das es einem User-Space-Thread gestattet, eine Mutex zu beanspruchen, ohne einen Kontextwechsel zum Kernel-Space zu benötigen, so lange die Mutex nicht bereits von einem anderen Thread gehalten wird.



Anmerkung

In diesem Dokument verwenden wir die Begriffe *Futex* und *Mutex*, um POSIX-Thread (pthread) - Mutex-Konstrukte zu beschreiben.

1. Wenn Sie ein **pthread_mutex_t**-Objekt mit den Standardattributen initialisieren, so wird eine private, nicht-rekursive, nicht-robuste und nicht-prioritätsverberende Mutex erstellt.
2. Unter pthreads können Mutexes mit folgender Zeichenfolge initialisiert werden:

```
pthread_mutex_t my_mutex;

pthread_mutex_init(&my_mutex, NULL);
```

3. In diesem Falle nutzt Ihre Applikation möglicherweise nicht die Vorteile der pthreads-Programmierschnittstelle und des MRG Realtime Kernels. Es gibt eine Reihe von Mutex-Optionen, die Sie in Erwägung ziehen sollten, wenn Sie eine Applikation scheiben oder portieren.

Fortgeschrittene Mutex-Optionen

Um zusätzliche Möglichkeiten für den Mutex zu definieren, werden Sie ein **pthread_mutexattr_t**-

Objekt erstellen müssen. Dieses Objekt speichert die definierten Attribute für den Futex.



Wichtig

Um die Beispiele kurz zu halten, enthalten sie keine Überprüfung des Wiedergabewertes der Funktion. Dies ist eine einfache Sicherheitsprozedur, die immer durchgeführt werden sollte.

1. Erstellen des Mutex-Objekts:

```
pthread_mutex_t my_mutex;

pthread_mutexattr_t my_mutex_attr;

pthread_mutexattr_init(&my_mutex_attr);
```

2. Gemeinsam verwendete und private Mutexe:

   Gemeinsam verwendete Mutexe können von mehreren Prozessen verwendet werden, allerdings können sie auch deutlich mehr Overhead verursachen.

```
pthread_mutexattr_setpshared(&my_mutex_attr, PTHREAD_PROCESS_SHARED);
```

3. Echtzeit-Prioritätsvererbung:

   Probleme mit Prioritätsinversion lassen sich durch Verwendung von Prioritätsvererbung vermeiden.

```
pthread_mutexattr_setprotocol(&my_mutex_attr, PTHREAD_PRIO_INHERIT);
```

4. Robuste Mutexe:

   Robuste Mutexe werden freigegeben, wenn der Besitzer erlischt, allerdings kann dies ebenfalls zu hohem Overhead führen. **_NP** in dieser Zeichenfolge zeigt an, dass diese Option nicht-POSIX oder nicht portabel ist.

```
pthread_mutexattr_setrobust_np(&my_mutex_attr, PTHREAD_MUTEX_ROBUST_NP);
```

5. Mutex-Initialisierung:

   SInd die Attribute gesetzt, initialisieren Sie ein Mutex unter Verwendung dieser Eigenschaften.

```
pthread_mutex_init(&my_mutex, &my_mutex_attr);
```

6. Bereinigung des Attributobjekts:

   Nachdem der Mutex erstellt wurde, können Sie das Attributobjekt behalten, um weitere Mutexe desselben Typs zu initialisieren oder aber Sie können es bereinigen. Weder das eine noch das andere wirkt sich auf den Mutex aus. Um das Attributobjekt zu bereinigen, verwenden Sie den **_destroy**-Befehl.

```
pthread_mutexattr_destroy(&my_mutex_attr);
```

   Der Mutex läuft nun als reguläre **pthread_mutex**, und kann ganz normal gesperrt, entsperrt und gelöscht werden.

Zugehörige Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- futex(7)
- pthread_mutex_destroy(P)

  Für Information zu **pthread_mutex_t** und **pthread_mutex_init**
- pthread_mutexattr_setprotocol(3p)

  Für Information zu **pthread_mutexattr_setprotocol** und **pthread_mutexattr_getprotocol**
- pthread_mutexattr_setprioceiling(3p)

  Für Information zu **pthread_mutexattr_setprioceiling** und **pthread_mutexattr_getprioceiling**

4.4. TCP_NODELAY und kleine Pufferschreibvorgänge

Wie kurz im Abschnitt Transmission Control Protocol (TCP) erläutert, verwendet TCP standardmäßig den Nagle-Algorithmus, um kleine, ausgehende Pakete zu sammeln und zusammen zu verschicken. Dies kann sich nachteilig auf die Latenz auswirken.

Verwendung von TCP_NODELAY und TCP_CORK zur Verbesserung von Netzwerklatenz

1. Applikationen, die eine niedrigere Latenz erfordern, sollten auf Sockets laufen, auf denen **TCP_NODELAY** aktiviert ist. Dies kann mithilfe des **setsockopt**-Befehls über die Socket-Programmierschnittstelle aktiviert werden:

   ```
   # int one = 1;

   # setsockopt(descriptor, SOL_TCP, TCP_NODELAY, &one, sizeof(one));
   ```

2. Um dies wirksam zu nutzen, müssen Applikationen kleine, logisch zusammengehörige Pufferschreibvorgänge vermeiden. Da **TCP_NODELAY** aktiviert ist, führen diese kleinen Schreibvorgänge dazu, dass TCP diese multiplen Puffer als einzelne Pakete verschickt, was zu einer verschlechterten Gesamtleistung führen kann.

   Wenn Applikationen mehrere logisch zusammengehörige Puffer besitzen, die als ein einzige Paket verschickt werden sollen, so wäre es möglich, im Speicher ein zusammenhängendes Paket zu erstellen und das logische Paket dann auf einem mit **TCP_NODELAY** konfigurierten Socket an TCP zu schicken.

   Alternativ erstellen Sie einen I/O-Vektor und geben ihn mittels **writev** auf einem mit **TCP_NODELAY** konfigurierten Socket an den Kernel.

3. Eine weitere Möglichkeit ist die Verwendung von **TCP_CORK**, was TCP mitteilt, dass auf die Entfernung der Sperre auf der Applikation gewartet werden soll, ehe weitere Pakete verschickt werden. Dieser Befehl führt dazu, dass empfangene Puffer an bestehende Puffer angehängt werden. Dies ermöglicht es Applikationen, ein Paket im Kernel-Space zu erstellen, das ggf. benötigt wird, wenn verschiedene Bibliotheken verwendet werden, die Abstraktionsschichten liefern. Um **TCP_CORK** zu aktivieren, setzen Sie es mithilfe der **setsockopt**-Socket-Programmierschnittstelle auf den Wert ***1***.

   ```
   # int one = 1;

   # setsockopt(descriptor, SOL_TCP, TCP_CORK, &one, sizeof(one));
   ```

4. Wenn das logische Paket im Kernel durch die verschiedenen Komponenten in der Applikation

erstellt wurde, teilen Sie TCP mit, die Sperre zu entfernen. TCP verschickt das akkumulierte logische Paket sofort, ohne dass auf weitere Pakete der Applikation gewartet wird.

```
# int zero = 0;

# setsockopt(descriptor, SOL_TCP, TCP_CORK, &zero, sizeof(zero));
```

Zugehörige Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- tcp(7)
- setsockopt(3p)
- setsockopt(2)

4.5. Einstellen von Echtzeit-Scheduler-Prioritäten

Die Verwendung von **`rtctl`** zur Einstellung von Scheduler-Prioritäten wird im Kapitel Verwendung des **`rtctl`**-Befehls zur Einstellung von Prioritäten beschrieben. In dem Beispiel in diesem Kapitel wurde einigen Kernel-Threads eine sehr hohe Priorität gegeben. Dadurch können die Standardprioritäten gut mit den Anforderungen der Realtime-Spezifikation für Java (RTSJ) integriert werden. RTSJ erfordert einen Prioritätsbereich von 10-89, daher werden die Prioritäten vieler Kernel-Threads bei 90 und darüber angesiedelt. Dies vermeidet unvorhersehbares Verhalten, wenn eine lang laufende Java-Applikation wesentliche Systemdienste am Laufen hindert.

Für Deployments, bei denen RTSJ nicht in Gebrauch ist, gibt es eine große Bandbreite an Scheduling-Prioritäten unter 90, die für Applikationen zur Verfügung stehen. Es ist in der Regel gefährlich, wenn Benutzerapplikationen auf Priorität 90 und darüber laufen - trotz der Tatsache, dass diese Möglichkeit besteht. Werden wichtige Systemdienste am Laufen gehindert, so kann es zu unvorhersehbarem Verhalten kommen, einschließlich blockiertem Netzwerkverkehr, blockierter Speicherauslagerung und Beschädigung von Daten aufgrund von blockiertem Dateisystem-Journaling.

Extreme Vorsicht sollte beim Scheduling von Applikations-Threads mit Priorität über 89 gelten. Besitzen Threads eine Priorität von über 89, so sollten Sie unbedingt sicherstellen, dass die Threads nur einen sehr kurzen Codepfad durchlaufen. Wird dies nicht beachtet, so würde dies die Niedriglatenz-Fähigkeiten des MRG Realtime Kernels untergraben.

Einstellung von Echtzeitpriorität für nicht-privilegierte Nutzer

Generell kann nur der Root-Benutzer die Prioritäts- und Scheduling-Informationen verändern. Sollen nicht-privilegierte Nutzer diese Einstellungen anpassen können, so ist die beste Methode, den Nutzer zur **`Realtime`**-Gruppe hinzuzufügen.



Wichtig

Sie können Benutzerprivilegien auch durch Bearbeitung der **`/etc/security/limits.conf`**-Datei ändern. Dies ist jedoch fehleranfällig und kann das System für reguläre Nutzer unbenutzbar machen. Falls Sie sich *für* die Bearbeitung dieser Datei entscheiden, seien Sie vorsichtig und erstellen Sie stets eine Sicherungskopie, ehe Sie Änderungen vornehmen.

Weitere Informationsquellen

Weitere Informationen diesbezüglich finden Sie unter den folgenden Links:

- Es gibt ein Test-Dienstprogramm namens **`signaltest`**, das nützlich zur Darstellung des MRG Realtime Systemverhaltens ist. Ein von Arnaldo Carvalho de Melo verfasstes Whitepaper erläutert dies detaillierter: signaltest: Using the RT priorities

4.6. Laden dynamischer Bibliotheken

Bei der Entwicklung Ihres MRG Realtime Programms sollten Sie die Auflösung von Symbolen beim Start erwägen. Obwohl dies die Programminitialisierung verlangsamen kann, bietet dies andererseits eine Möglichkeit, nicht-deterministische Latenzen während der Programmausführung zu vermeiden.

Dynamische Bibliotheken können angewiesen werden, beim Systemstart zu laden, indem die **`LD_BIND_NOW`**-Variable mit **`ld.so`**, dem dynamischen Linker/Lader, eingestellt wird.

Sehen Sie nachfolgend ein Beispiel für ein Shell-Skript. Dieses Skript exportiert die **`LD_BIND_NOW`**-Variable mit einem nicht-Null Wert von ***1*** und führt dann ein Programm mit der Scheduling-Richtlinie FIFO und der Priorität ***1*** aus.

```
#!/bin/sh

LD_BIND_NOW=1
export LD_BIND_NOW

chrt --fifo 1 /opt/myapp/myapp-server &
```

Zugehörige Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten:

- ld.so(8)

4.7. Verwenden der _COARSE POSIX-Uhren für Applikations-Timestamps

Applikationen, die regelmäßig Timestamps durchführen, haben den Aufwand zu tragen, der durch das Lesen der Systemuhr entsteht. Ist der Aufwand und die Zeit, die zum Lesen der Uhr aufgewendet wird, zu hoch, kann sich dies negativ auf die Leistung der Applikation auswirken.

Stellen Sie sich zur Veranschaulichung eine Uhr vor, die in einer Schublade aufbewahrt wird. Sie müssen auf diese Uhr sehen, um die Zeit für bestimmte Ereignisse zu messen, die Sie beobachten. Jedes Mal müssen Sie die Schublade öffnen und die Uhr hervorholen, um die Zeit ablesen zu können. Der Aufwand zum Ablesen der Zeit ist zu hoch und kann dazu führen, dass Sie Ereignisse verpassen oder diese mit fehlerhaften Zeitangaben versehen.

Im Gegensatz dazu könnten Sie eine Wanduhr schneller ablesen, was die Zeitmessung der beobachteten Ergebnisse weniger behindern würde. Stehen Sie vor der Wand, ist das Ablesen der Zeit noch schneller.

Ganz ähnlich funktioniert diese Leistungsoptimierung (durch Reduktion des Aufwands zum Ablesen der Uhr), indem eine Hardware-Uhr gewählt wird, die einen schnelleren Ablesemechanismus hat. In MRG Realtime kann die Leistung weiter verbessert werden, indem POSIX-Uhren mit der **`clock_gettime()`**-Funktion verwendet werden, um den geringst möglichen Aufwand zum Ablesen der Uhr zu haben.

POSIX-Uhren

POSIX umfasst einen Standard zur Implementierung und Darstellung von Zeitquellen. Die POSIX-Uhr kann von jeder Applikation separat ausgewählt werden, ohne Auswirkungen auf andere Applikationen im System. Dies unterscheidet sie von der Hardware-Uhr, die in Abschnitt 2.6, „Verwenden von Hardware-Uhren für System-Timestamps“ beschrieben wird, die vom Kernel ausgewählt und systemweit implementiert wird.

Die Funktion zum Ablesen einer POSIX-Uhr ist **`clock_gettime()`**, definiert unter **`<time.h>`**. **`clock_gettime()`** hat eine Entsprechung im Kernel in Form eines Systemaufrufs. Wenn der Benutzerprozess **`clock_gettime()`** aufruft, dann ruft die entsprechende C-Bibliothek (**`glibc`**) den **`sys_clock_gettime()`**-Systemaufruf auf, der die angeforderte Operation ausführt und das Ergebnis an das Benutzerprogramm zurückgibt.

Dieser Kontextwechsel von der Benutzerapplikation zum Kernel ist jedoch mit Aufwand verbunden. Zwar ist dieser Aufwand an sich gering; falls die Operation jedoch tausendfach wiederholt wird, kann sich der kumulierte Aufwand auf die Leistung der Applikation insgesamt auswirken. Um diesen Kontextwechsel zum Kernel zu vermeiden und somit das Ablesen der Uhr zu beschleunigen, wurde Unterstützung für die **`CLOCK_MONOTONIC_COARSE`** und **`CLOCK_REALTIME_COARSE`** POSIX-Uhren in Form einer VDSO-Bibliotheksfunktion hinzugefügt.

Das Ablesen der Zeit durch **`clock_gettime()`** mithilfe einer der _***COARSE*** Uhr-Varianten erfordert kein Eingreifen des Kernels und erfolgt gänzlich im User Space, wodurch die Leistung deutlich verbessert wird. Die von _***COARSE***-Uhren abgelesenen Zeiten haben eine Auflösung von einer Millisekunde (ms), was bedeutet, dass Zeitabstände kleiner als 1 ms nicht aufgezeichnet werden. Die _***COARSE***-Varianten der POSIX-Uhren sind geeignet für Applikationen, denen eine Millisekunden-Auflösung reicht, und die Vorteile sind auf jenen Systemen deutlicher, die bislang Hardware-Uhren mit hohem Aufwand zum Ablesen verwendeten.



Anmerkung

Werfen Sie einen Blick auf das *MRG Realtime Referenzhandbuch* für einen Vergleich des Aufwands und der Auflösung für POSIX-Uhren mit bzw. ohne das _***COARSE***-Präfix.

Ältere MRG Realtime Kernel boten eine systemweite Timestamp-Funktion. Diese erforderte zwar keine Änderungen in der Applikation, wirkte sich jedoch auf alle Applikationen auf dem System aus. Der aktuelle MRG Realtime Kernel bietet eine besser steuerbare Lösung, die es jeder Applikation einzeln erlaubt, eine POSIX-Uhr für optimale Leistung zu wählen, ohne Auswirkungen auf andere Applikationen. In der Regel muss hierzu lediglich in den **`clock_gettime()`**-Aufrufen im Quellcode **`CLOCK_MONOTONIC`** durch **`CLOCK_MONOTONIC_COARSE`** ersetzt werden, zum Beispiel:

Beispiel 4.1. Verwenden der *_COARSE*-Uhrvariante in `clock_gettime`

```
#include <time.h>

main()
{
 int rc;
 long i;
 struct timespec ts;

 for(i=0; i<10000000; i++) {
  rc = clock_gettime(CLOCK_MONOTONIC_COARSE, &ts);
 }
}
```

Sie können das obige Beispiel weiter verbessern, indem Sie beispielsweise Funktionen einfügen, um den Wiedergabewert von **`clock_gettime()`** zu prüfen, um den Wert der ***`rc`***-Variable zu prüfen, oder um zu gewährleisten, dass dem Inhalt der ***`ts`***-Struktur vertraut werden kann. Auf der **`clock_gettime()`**-Handbuchseite finden Sie weitere Informationen über das Schreiben zuverlässigerer Applikationen.



Wichtig

Programme, die die **`clock_gettime()`**-Funktion nutzen, müssen mit der **`rt`**-Bibliothek verknüpft werden, indem ***`'-lrt'`*** zur **`gcc`**-Befehlszeile hinzugefügt wird.

```
cc clock_timing.c -o clock_timing -lrt
```

Zugehörige Handbuchseiten

Weitere Informationen diesbezüglich finden Sie auf den folgenden Handbuchseiten bzw. in den folgenden Büchern:

- clock_gettime()
- *Linux System Programming* von Robert Love
- *Understanding The Linux Kernel* von Daniel P. Bovet und Marco Cesati

Kapitel 5. Weitere Informationen

5.1. Melden von Fehlern

Fehlerdiagnose

Bevor Sie eine Fehlermeldung erstatten, befolgen Sie diese Schritte, um festzustellen, wo die eigentliche Fehlerursache liegt. Dies ist überaus hilfreich bei der Fehlerbehebung.

1. Stellen Sie sicher, dass Sie die aktuellste Version des Red Hat Enterprise Linux 6 Kernels verwenden, und booten Sie diesen vom "grub"-Menü. Versuchen Sie dann, das Problem mit dem Standard-Kernel zu reproduzieren. Falls das Problem nach wie vor auftritt, melden Sie den Fehler für Red Hat Enterprise Linux 6, *NICHT* für MRG Realtime.
2. Falls das Problem bei Verwendung des Standard-Kernels nicht auftritt, so ist der Fehler wahrscheinlich ein Ergebnis späterer Änderungen in entweder:
    a. dem Upstream-Kernel, auf dem MRG Realtime basiert. Zum Beispiel basiert der Red Hat Enterprise Linux 6 Kernel auf 2.6.32 und MRG Realtime basiert auf Version 3.2
    b. MRG Realtime spezifischen Erweiterungen, die Red Hat zusätzlich zum Grund-Kernel (3.2) angewendet hat

    Um das Problem näher zu bestimmen, versuchen Sie, das Problem an einem unbearbeiteten Upstream 3.2 Kernel reprodzieren können. Aus diesem Grund liefern wir neben dem MRG Realtime Kernel auch eine **`vanilla`**-Kernelvariante. Beim **`vanilla`**-Kernel handelt es sich um einen Upstream Kernel-Build ohne die MRG Realtime Features.

Melden eines Fehlers

Wenn Sie sicher sind, dass der Fehler spezifisch für MRG Realtime ist, folgen Sie diesen Anweisungen, um den Fehler zu melden:

1. Erstellen Sie einen Bugzilla-Benutzerkonto.
2. Melden Sie sich an und klicken Sie auf Enter A New Bug Report.
3. Sie müssen nun das Produkt angeben, in dem der Fehler auftritt. MRG Realtime erscheint in der Red Hat Produktliste unter Red Hat Enterprise MRG. Es ist wichtig, dass Sie das richtige Produkt angeben, in dem der Fehler auftritt.
4. Fahren Sie mit der Eingabe der Fehlerinformationen fort, indem Sie die entsprechende Komponente zuordnen und eine ausführliche Problembeschreibung liefern. Bei Eingabe der Problembeschreibung sollten Sie auch mitteilen, ob Sie das Problem mit dem standardmäßigen Red Hat Enterprise Linux 6 oder dem bereitgestellten **`vanilla`**-Kernel reproduzieren konnten.

5.2. Weitere Informationsquellen

- Red Hat Enterprise MRG Produktinformationen
    - http://www.redhat.com/mrg
- MRG Realtime Installationshandbuch und andere Red Hat Enterprise MRG Dokumentation
    - http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_MRG/
- Mailing-Liste
    - Um auf dieser Liste zu posten, schicken Sie eine E-Mail an rhemrg-users-list@redhat.com
    - Die Mailing-Liste können Sie unter https://www.redhat.com/mailman/listinfo/rhemrg-users-list abonnieren.

Ereignis-Tracing

```
    Event Tracing

  Documentation written by Theodore Ts'o
  Updated by Li Zefan and Tom Zanussi

1. Introduction
===============

Tracepoints (see Documentation/trace/tracepoints.txt) can be used
without creating custom kernel modules to register probe functions
using the event tracing infrastructure.

Not all tracepoints can be traced using the event tracing system;
the kernel developer must provide code snippets which define how the
tracing information is saved into the tracing buffer, and how the
tracing information should be printed.

2. Using Event Tracing
======================

2.1 Via the 'set_event' interface
---------------------------------

The events which are available for tracing can be found in the file
/sys/kernel/debug/tracing/available_events.

To enable a particular event, such as 'sched_wakeup', simply echo it
to /sys/kernel/debug/tracing/set_event. For example:

 # echo sched_wakeup >> /sys/kernel/debug/tracing/set_event

[ Note: '>>' is necessary, otherwise it will firstly disable
  all the events. ]

To disable an event, echo the event name to the set_event file prefixed
with an exclamation point:

 # echo '!sched_wakeup' >> /sys/kernel/debug/tracing/set_event

To disable all events, echo an empty line to the set_event file:

 # echo > /sys/kernel/debug/tracing/set_event

To enable all events, echo '*:*' or '*:' to the set_event file:

 # echo *:* > /sys/kernel/debug/tracing/set_event

The events are organized into subsystems, such as ext4, irq, sched,
etc., and a full event name looks like this: <subsystem>:<event>.  The
subsystem name is optional, but it is displayed in the available_events
file.  All of the events in a subsystem can be specified via the syntax
"<subsystem>:*"; for example, to enable all irq events, you can use the
command:

 # echo 'irq:*' > /sys/kernel/debug/tracing/set_event

2.2 Via the 'enable' toggle
---------------------------

The events available are also listed in /sys/kernel/debug/tracing/events/ hierarchy
```

```
of directories.

To enable event 'sched_wakeup':

 # echo 1 > /sys/kernel/debug/tracing/events/sched/sched_wakeup/enable

To disable it:

 # echo 0 > /sys/kernel/debug/tracing/events/sched/sched_wakeup/enable

To enable all events in sched subsystem:

 # echo 1 > /sys/kernel/debug/tracing/events/sched/enable

To enable all events:

 # echo 1 > /sys/kernel/debug/tracing/events/enable

When reading one of these enable files, there are four results:

 0 - all events this file affects are disabled
 1 - all events this file affects are enabled
 X - there is a mixture of events enabled and disabled
 ? - this file does not affect any event

2.3 Boot option
---------------

In order to facilitate early boot debugging, use boot option:

 trace_event=[event-list]

The format of this boot option is the same as described in section 2.1.

3. Defining an event-enabled tracepoint
========================================

See The example provided in samples/trace_events

4. Event formats
================

Each trace event has a 'format' file associated with it that contains
a description of each field in a logged event.  This information can
be used to parse the binary trace stream, and is also the place to
find the field names that can be used in event filters (see section 5).

It also displays the format string that will be used to print the
event in text mode, along with the event name and ID used for
profiling.

Every event has a set of 'common' fields associated with it; these are
the fields prefixed with 'common_'.  The other fields vary between
events and correspond to the fields defined in the TRACE_EVENT
definition for that event.

Each field in the format has the form:

     field:field-type field-name; offset:N; size:N; signed:N;
```

```
where offset is the offset of the field in the trace record and size
is the size of the data item, in bytes, signed will be 0 or 1 denoting
if the type of field is signed or not.

For example, here's the information displayed for the 'sched_wakeup'
event:

# cat /sys/kernel/debug/tracing/events/sched/sched_wakeup/format
name: sched_wakeup
ID: 62
format:
 field:unsigned short common_type; offset:0; size:2; signed:0;
 field:unsigned char common_flags; offset:2; size:1; signed:0;
 field:unsigned char common_preempt_count; offset:3; size:1; signed:0;
 field:int common_pid; offset:4; size:4; signed:1;
 field:int common_lock_depth; offset:8; size:4; signed:1;

 field:char comm[TASK_COMM_LEN]; offset:12; size:16; signed:1;
 field:pid_t pid; offset:28; size:4; signed:1;
 field:int prio; offset:32; size:4; signed:1;
 field:int success; offset:36; size:4; signed:1;
 field:int target_cpu; offset:40; size:4; signed:1;

print fmt: "comm=%s pid=%d prio=%d success=%d target_cpu=%03d", REC->comm, REC-
>pid, REC->prio, REC->success, REC->target_cpu

This event contains 10 fields, the first 5 common and the remaining 5
event-specific.  All the fields for this event are numeric, except for
'comm' which is a string, a distinction important for event filtering.

5. Event filtering
==================

Trace events can be filtered in the kernel by associating boolean
'filter expressions' with them.  As soon as an event is logged into
the trace buffer, its fields are checked against the filter expression
associated with that event type.  An event with field values that
'match' the filter will appear in the trace output, and an event whose
values don't match will be discarded.  An event with no filter
associated with it matches everything, and is the default when no
filter has been set for an event.

5.1 Expression syntax
---------------------

A filter expression consists of one or more 'predicates' that can be
combined using the logical operators '&&' and '||'.  A predicate is
simply a clause that compares the value of a field contained within a
logged event with a constant value and returns either 0 or 1 depending
on whether the field value matched (1) or didn't match (0):

   field-name relational-operator value

Parentheses can be used to provide arbitrary logical groupings and
double-quotes can be used to prevent the shell from interpreting
operators as shell meta characters.

The field-names available for use in filters can be found in the
'format' files for trace events (see section 4).
```

```
The relational-operators depend on the type of the field being tested:

The operators available for numeric fields are:

==, !=, <, <=, >, >=

And for string fields they are:

==, !=

Currently, only exact string matches are supported.

Currently, the maximum number of predicates in a filter is 16.

5.2 Setting filters
-------------------

A filter for an individual event is set by writing a filter expression
to the 'filter' file for the given event.

For example:

# cd /sys/kernel/debug/tracing/events/sched/sched_wakeup
# echo "common_preempt_count > 4" > filter

A slightly more involved example:

# cd /sys/kernel/debug/tracing/events/signal/signal_generate
# echo "((sig >= 10 && sig < 15) || sig == 17) && comm != bash" > filter

If there is an error in the expression, you'll get an 'Invalid
argument' error when setting it, and the erroneous string along with
an error message can be seen by looking at the filter e.g.:

# cd /sys/kernel/debug/tracing/events/signal/signal_generate
# echo "((sig >= 10 && sig < 15) || dsig == 17) && comm != bash" > filter
-bash: echo: write error: Invalid argument
# cat filter
((sig >= 10 && sig < 15) || dsig == 17) && comm != bash
^
parse_error: Field not found

Currently the caret ('^') for an error always appears at the beginning of
the filter string; the error message should still be useful though
even without more accurate position info.

5.3 Clearing filters
--------------------

To clear the filter for an event, write a '0' to the event's filter
file.

To clear the filters for all events in a subsystem, write a '0' to the
subsystem's filter file.

5.3 Subsystem filters
---------------------

For convenience, filters for every event in a subsystem can be set or
cleared as a group by writing a filter expression into the filter file
```

```
at the root of the subsystem.  Note however, that if a filter for any
event within the subsystem lacks a field specified in the subsystem
filter, or if the filter can't be applied for any other reason, the
filter for that event will retain its previous setting.  This can
result in an unintended mixture of filters which could lead to
confusing (to the user who might think different filters are in
effect) trace output.  Only filters that reference just the common
fields can be guaranteed to propagate successfully to all events.

Here are a few subsystem filter examples that also illustrate the
above points:

Clear the filters on all events in the sched subsytem:

# cd /sys/kernel/debug/tracing/events/sched
# echo 0 > filter
# cat sched_switch/filter
none
# cat sched_wakeup/filter
none

Set a filter using only common fields for all events in the sched
subsytem (all events end up with the same filter):

# cd /sys/kernel/debug/tracing/events/sched
# echo common_pid == 0 > filter
# cat sched_switch/filter
common_pid == 0
# cat sched_wakeup/filter
common_pid == 0

Attempt to set a filter using a non-common field for all events in the
sched subsytem (all events but those that have a prev_pid field retain
their old filters):

# cd /sys/kernel/debug/tracing/events/sched
# echo prev_pid == 0 > filter
# cat sched_switch/filter
prev_pid == 0
# cat sched_wakeup/filter
common_pid == 0
```

Funktions-Tracer

```
  ftrace - Function Tracer
========================

Copyright 2012 Red Hat Inc.
   Author:   Steven Rostedt <srostedt@redhat.com>
  License:   The GNU Free Documentation License, Version 1.2
                (dual licensed under the GPL v2)
Reviewers:   Elias Oltmanns, Randy Dunlap, Andrew Morton,
      John Kacur, and David Teigland.
Written for: 3.2.16-rt27-mrg

Introduction
------------

Ftrace is an internal tracer designed to help out developers and
designers of systems to find what is going on inside the kernel.
It can be used for debugging or analyzing latencies and
performance issues that take place outside of user-space.

Although ftrace is the function tracer, it also includes an
infrastructure that allows for other types of tracing. Some of
the tracers that are currently in ftrace include a tracer to
trace the time it takes for a high priority task to run after it
was woken up, the time interrupts are disabled.

Static trace event points are also spread throughout the kernel.
These trace events can be enabled to show specific information about
a part of the kernel (like context switches, system calls, interrupts,
memory, etc). The nice thing about the trace events is that they show
up in all tracers and even the nop (off) tracer.


Implementation Details
----------------------

See ftrace-design.txt for details for arch porters and such.


The File System
---------------

Ftrace uses the debugfs file system to hold the control files as
well as the files to display output.

When debugfs is configured into the kernel (which selecting any ftrace
option will do) the directory /sys/kernel/debug will be created. To mount
this directory, you can add to your /etc/fstab file:

 debugfs       /sys/kernel/debug          debugfs defaults        0       0

Or you can mount it at run time with:

 mount -t debugfs nodev /sys/kernel/debug

For quicker access to that directory you may want to make a soft link to
it:

 ln -s /sys/kernel/debug /debug

Any selected ftrace option will also create a directory called tracing
```

```
within the debugfs. The rest of the document will assume that you are in
the ftrace directory (cd /sys/kernel/debug/tracing) and will only concentrate
on the files within that directory and not distract from the content with
the extended "/sys/kernel/debug/tracing" path name.

That's it! (assuming that you have ftrace configured into your kernel)

After mounting the debugfs, you can see a directory called
"tracing".  This directory contains the control and output files
of ftrace. Here is a list of some of the key files:


 Note: all time values are in microseconds.

  current_tracer:

 This is used to set or display the current tracer
 that is configured.

  available_tracers:

 This holds the different types of tracers that
 have been compiled into the kernel. The
 tracers listed here can be configured by
 echoing their name into current_tracer.

  trace:

 This file holds the output of the trace in a human
 readable format (described below). Note, tracing will
 be temporarily disabled while this file is read. The
 "trace" file is static, and if the tracer is not
 adding more data, it will display the same
 information every time it is read.

  trace_pipe:

 The output is the same as the "trace" file but this
 file is meant to be streamed with live tracing.
 Reads from this file will block until new data is
 retrieved.  Unlike the "trace" file, this file is a
 consumer. This means reading from this file causes
 sequential reads to display more current data. Once
 data is read from this file, it is consumed, and
 will not be read again with a sequential read.

        This file will not disable tracing when read, like the
        "trace" file does.

  trace_options:

 This file lets the user control the amount of data
 that is displayed in one of the above output
 files. Some of the options even modify the behavior
 of the trace.

  tracing_max_latency:

 Some of the tracers record the max latency.
 For example, the time interrupts are disabled.
```

```
This time is saved in this file. The max trace
will also be stored, and displayed by "trace".
A new max trace will only be recorded if the
latency is greater than the value in this
file. (in microseconds)

By writing an ASCII '0' into this file, it will
reset the max latency and the next latency will be
recorded. Writing a ASCII number other than zero
(ie. "123") will set the max latency to that number
and the next latency to be recorded will have to be
greater than the number in tracing_max_latency.

 buffer_size_kb:

This sets or displays the number of kilobytes each CPU
buffer can hold. The tracer buffers are the same size
for each CPU. The displayed number is the size of the
CPU buffer and not the total size of all buffers. The
trace buffers are allocated in pages (blocks of memory
that the kernel uses for allocation, usually 4 KB in size).
If the last page allocated has room for more bytes
than requested, the rest of the page will be used,
making the actual allocation bigger than requested.
( Note, the size may not be a multiple of the page size
  due to buffer management overhead. )

 buffer_total_size_kb

        This shows the total size of all allocated buffers.
        The buffer_size_kb shows the size of each individual CPU
        buffer. To know the full buffer size of all the individual
        CPU buffers combined, view this file.

 tracing_cpumask:

This is a mask that lets the user only trace
on specified CPUS. The format is a hex string
representing the CPUS.

 set_ftrace_filter:

When dynamic ftrace is configured in (see the
section below "dynamic ftrace"), the code is dynamically
modified (code text rewrite) to disable calling of the
function profiler (mcount). This lets tracing be configured
in with practically no overhead in performance.  This also
has a side effect of enabling or disabling specific functions
to be traced. Echoing names of functions into this file
will limit the trace to only those functions.

 set_ftrace_notrace:

This has an effect opposite to that of
set_ftrace_filter. Any function that is added here will not
be traced. If a function exists in both set_ftrace_filter
and set_ftrace_notrace, the function will _not_ be traced.

 set_ftrace_pid:
```

```
 Have the function tracer only trace a single thread.

  set_graph_function:

 Set a "trigger" function where tracing should start
 with the function graph tracer (See the section
 "dynamic ftrace" for more details). The functions here
        will make the function_graph tracer show just what these
        functions call (max of 32 functions can be added here).

  available_filter_functions:

 This lists the functions that ftrace
 has processed and can trace. These are the function
 names that you can pass to "set_ftrace_filter" or
 "set_ftrace_notrace". (See the section "dynamic ftrace"
 below for more details.)


The Tracers
-----------

Here is the list of current tracers that may be configured.

  "function"

 Function call tracer to trace all kernel functions.

  "function_graph"

 Similar to the function tracer except that the
 function graph tracer traces both the entry and exit
 of each function. It then provides the ability to draw
 a graph of the function calls similar to what C code
 source would look like, as well as the time spent in
 that particular function. Note, the time of the function
 will include the overhead of the tracer if that function
 called other functions that were traced.

  "irqsoff"

 Traces the areas that disable interrupts and saves
 the trace with the longest max latency.
 See tracing_max_latency. When a new max is recorded,
 it replaces the old trace. It is best to view this
 trace with the latency-format option enabled.

 (Note latency-format is automatically enabled when
  "irqsoff" tracer is enabled.)

  "wakeup_rt"

        Traces and records the max latency that it takes for
        the highest priority task to get scheduled after
        it has been woken up. This differs from the "wakeup" tracer
        as it only considers tasks with a real-time priority.
        As non-real-time tasks may take longer to wake up due
        to fair balance scheduling, they can hide a long latency
        of a real-time task. Use this tracer if you are only
        concerned about the wake up latency of real-time tasks.
```

```
        (Note latency-format is automatically enabled when
         "wakeup_rt" tracer is enabled.)

  "preemptoff"

 Similar to irqsoff but traces and records the amount of
 time for which preemption is disabled.

 (Note latency-format is automatically enabled when
  "preemptsoff" tracer is enabled.)

  "preemptirqsoff"

 Similar to irqsoff and preemptoff, but traces and
 records the largest time for which irqs and/or preemption
 is disabled.

 (Note latency-format is automatically enabled when
  "preemptirqsoff" tracer is enabled.)

  "wakeup"

 Traces and records the max latency that it takes for
 the highest priority task to get scheduled after
 it has been woken up.

 (Note latency-format is automatically enabled when
  "wakeup" tracer is enabled.)

  "nop"

 This is the "trace nothing" tracer. To remove all
 tracers from tracing simply echo "nop" into
 current_tracer.

 Note, this is also useful to view only trace events.

Trace Events
------------

Along with the tracers, there are trace events that are static points
within the kernel that can be enabled or disabled to trace. When
an event is enabled, it will be recorded within the recording
of the tracer. All tracers can view trace events.

For more information about trace events, see:

  Documentation/trace/events.txt


Examples of using the tracer
----------------------------

Here are typical examples of using the tracers when controlling
them only with the debugfs interface (without using any
user-land utilities).

Output format:
--------------
```

```
Here is an example of the output format of the file "trace"

                              --------
# tracer: function
#
#            TASK-PID    CPU#    TIMESTAMP  FUNCTION
#               | |       |          |         |
             bash-2030  [001] 14035.027994: prefetchw <-__kmalloc
             bash-2030  [001] 14035.027997: alloc_fdmem <-alloc_fdtable
             bash-2030  [001] 14035.027997: __kmalloc <-alloc_fdmem
             bash-2030  [001] 14035.027998: __find_general_cachep <-__kmalloc
                              --------

A header is printed with the tracer name that is represented by
the trace. In this case the tracer is "function". Then a header
showing the format. Task name "bash", the task PID "2030", the
CPU that it was running on "001", the timestamp in <secs>.<usecs>
format, the function name that was traced "prefetchw" and the
parent function that called this function "__kmalloc". The
timestamp is the time at which the function was entered.

The prio is the internal kernel priority, which is the inverse
of the priority that is usually displayed by user-space tools.
Zero represents the highest priority (99). Prio 100 starts the
"nice" priorities with 100 being equal to nice -20 and 139 being
nice 19.


Latency trace format
--------------------

When the latency-format option is enabled, the trace file gives
somewhat more information to see why a latency happened.
Some tracers automatically enable the latency format, but you can enable
or disable it for any tracer:

Enabling:
  # echo latency-format > trace_options
 or
  # echo 1 > options/latency-format

Disabling:
  # echo nolatency-format > trace_options
 or
  # echo 0 > options/latency-format

Here is a typical trace.

# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.2.16-rt27
# --------------------------------------------------------------------
# latency: 49 us, #130/130, CPU#2 | (Mreempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: swapper/2-0 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: __schedule
#  => ended at:   finish_task_switch
#
```

```
#
#                    _------=> CPU#
#                   / _-----=> irqs-off
#                  | / _----=> need-resched
#                  || / _---=> hardirq/softirq
#                  ||| / _--=> preempt-depth
#                  |||| / _--=> migrate-disable
#                  ||||| /     delay
#  cmd     pid    |||||| time  |   caller
#     \   /        |||||  \   |   /
ksoftirq-16      2d..1.    0us : _raw_spin_lock_irq <__schedule
ksoftirq-16      2d..1.    0us : add_preempt_count <_raw_spin_lock_irq
ksoftirq-16      2d..2.    0us : do_raw_spin_lock <_raw_spin_lock_irq
ksoftirq-16      2d..2.    1us : signal_pending_state <__schedule


This shows that the current tracer is "irqsoff" tracing the time
for which interrupts were disabled. It gives the trace version
and the version of the kernel upon which this was executed on
(3.2.16-rt27). Then it displays the max latency in microsecs (97
us). The number of trace entries displayed and the total number
recorded (both are three: #130/130). The type of preemption that was
used (preempt). VP, KP, SP, and HP are always zero and are
reserved for later use. #P is the number of online CPUS (#P:4).

The task is the process that was running when the latency
occurred. (swapper/2 pid: 0).

The start and stop (the functions in which the interrupts were
disabled and enabled respectively) that caused the latencies:

  __schedule is where the interrupts were disabled.
  finish_task_switch is where they were enabled again.

The next lines after the header are the trace itself. The header
explains which is which.

  cmd: The name of the process in the trace.

  pid: The PID of that process.

  CPU#: The CPU which the process was running on.

  irqs-off: 'd' interrupts are disabled. '.' otherwise.
     Note: If the architecture does not support a way to
    read the irq flags variable, an 'X' will always
    be printed here.

  need-resched: 'N' task need_resched is set, '.' otherwise.

  hardirq/softirq:
 'H' - hard irq occurred inside a softirq.
 'h' - hard irq is running
 's' - soft irq is running
 '.' - normal context.

  preempt-depth: The level of preempt_disabled

  migrate-disable: for the real-time kernel, tasks can be temporarially
            bound to a CPU. When this occurs, the tasks migrate-disable
```

```
            count is incremented. This will show a number if migrate-disable
            is set to something other than zero, and a '.' if it is zero.

The above is mostly meaningful for kernel developers.

  time: When the latency-format option is enabled, the trace file
 output includes a timestamp relative to the start of the
 trace. This differs from the output when latency-format
 is disabled, which includes an absolute timestamp from boot up.

  delay: This is just to help catch your eye a bit better. And
  needs to be fixed to be only relative to the same CPU
  (but doesn't matter for (preempt)(irqs)off tracers as
  they are for single CPUs anyway).
  The marks are determined by the difference between this
  current trace event and the next trace event.
   '!' - greater than preempt_mark_thresh (default 100)
   '+' - greater than 1 microsecond
   ' ' - less than or equal to 1 microsecond.

  The rest is the same as the 'trace' file.


trace_options
-------------

The trace_options file is used to control what gets printed in
the trace output. To see what is available, simply cat the file:

  cat trace_options
   print-parent
   nosym-offset
   nosym-addr
   noverbose
   noraw
   nohex
   nobin
   noblock
   nostacktrace
   trace_printk
   noftrace_preempt
   nobranch
   annotate
   nouserstacktrace
   nosym-userobj
   noprintk-msg-only
   context-info
   latency-format
   sleep-time
   graph-time
   record-cmd
   overwrite
   nodisable_on_free

Some options appear only when a tracer is active:

  [blk]
   noblk_classic

  [function]
```

```
    nofunc_stack_trace

  [function_graph]
    nofuncgraph-overrun
    funcgraph-cpu
    funcgraph-overhead
    nofuncgraph-proc
    funcgraph-duration
    nofuncgraph-abstime
    funcgraph-irqs

  [wakeup, wakeup_rt, irqsoff, preemptoff, preemptirqsoff]
    nodisplay-graph

To disable one of the options, echo in the option prepended with
"no".

  echo noprint-parent > trace_options

To enable an option, leave off the "no".

  echo sym-offset > trace_options

Each of these options also exist in the options directory, and can
be enabled and disabled by writing in an ASCII '1' or '0' respectively.

    echo 0 > options/print-parent
    echo 1 > options/sym-offset

Here are the available options:

  print-parent - On function traces, display the calling (parent)
    function as well as the function being traced.

  print-parent:
    bash-4000  [01]  1477.606694: simple_strtoul <-strict_strtoul

  noprint-parent:
    bash-4000  [01]  1477.606694: simple_strtoul


  sym-offset - Display not only the function name, but also the
          offset in the function. For example, instead of
          seeing just "ktime_get", you will see
          "ktime_get+0xb/0x20".

  sym-offset:
    bash-4000  [01]  1477.606694: simple_strtoul+0x6/0xa0

  sym-addr - this will also display the function address as well
        as the function name.

  sym-addr:
    bash-4000  [01]  1477.606694: simple_strtoul <c0339346>

  verbose - This deals with the trace file when the
            latency-format option is enabled.

     bash  4000 1 0 00000000 00010a95 [58127d26] 1720.415ms \
     (+0.000ms): simple_strtoul (strict_strtoul)
```

```
  raw - This will display raw numbers. This option is best for
 use with user applications that can translate the raw
 numbers better than having it done in the kernel.

  hex - Similar to raw, but the numbers will be in a hexadecimal
 format.

  bin - This will print out the formats in raw binary numbers
        It is still ASCII, just not human readable.

  block - deprecated

  stacktrace - This is one of the options that changes the trace
        itself. When a trace is recorded, so is the stack
        of functions. This allows for back traces of
        trace sites. This does not affect the function or
        function graph events.

  userstacktrace - This option changes the trace. It records a
     stacktrace of the current userspace thread.
     Note, this will go to the user space function
     and depending if the user space application
     was compiled with frame pointers, it may or
     may not go deeper into the user space call stack.

  sym-userobj - when user stacktrace are enabled, look up which
  object the address belongs to, and print a
  relative address. This is especially useful when
  ASLR is on, otherwise you don't get a chance to
  resolve the address to object/file/line after
  the app is no longer running

  The lookup is performed when you read
  trace,trace_pipe. Example:

  a.out-1623  [000] 40874.465068: /root/a.out[+0x480] <-/root/a.out[+0
x494] <- /root/a.out[+0x4a8] <- /lib/libc-2.7.so[+0x1e1a6]

  context-info - This prints out the prefix to most events. When disabled
          only the content of the event is shown.

     content-info:

    <idle>-0  [000] 63974.137819:    0:120:R   + [000]     5: 50:S sirq-timer/0

     nocontent-info:

     0:120:R   + [000]     5: 50:S sirq-timer/0


  trace_printk - When the kernel has trace_printk()s used, this option
              can disable them. Otherwise they always write into the
   ring buffer.

  printk-msg-only - When trace_printk()s are used in the kernel,
        sometimes only the printk message is desired.
      Like nocontext-info, enabling printk-msg-only
      will remove all context from trace_printks only.
```

```
For a - trace_printk("jiffies are %ld\n", jiffies);

  noprintk-msg-only:

<...>-2866  [003] 24152.494117: ftrace_print_test: jiffies are 4318859691

  printk-msg-only:

 jiffies are 4318859691


ftrace_preempt - When the function tracer is running, it disables
      interrupts while it records its trace. Enabling
   ftrace_preempt, will make the function trace
   only disable preemption. But because function
   tracing must disable tracing to prevent recursion
   this may miss tracing interrupts that happen while
   a function was being traced.

branch - When the branch tracer is configured, by enabling the
    branch option, all locations that have likely() and
  unlikely() branch annotations in the kernel, will be
  traced. Note, this has very high overhead.

annotate - Because the ring buffer is split into per cpu buffers,
      to prevent confusion about when a CPU buffer starts
    compared to the other CPU buffers, an annotation is
    displayed. This option is can disable the annotation.

      ##### CPU 1 buffer started ####

sched-tree - trace all tasks that are on the runqueue, at
      every scheduling event. Will add overhead if
      there's a lot of tasks running at once.

latency-format - This option changes the trace. When
                 it is enabled, the trace displays
                 additional information about the
                 latencies, as described in "Latency
                 trace format".

sleep-time - When the function graph tracer is running, the time
        it schedules out is also recorded. When the task
      schedules back in, the time it scheduled out is
      also included in the time of the function.
      When "sleep-time" is disabled, the time a task is
      scheduled out is not included.

   sleep-time:

        0) ! 388.106 us  |  }

   nosleep-time:

        2) + 13.116 us   |  }

graph-time - This is used with the function profiler when function
             graph tracing is enabled. The timing for functions by
             default will be the entire time a function is running
             including all the functions it calls. If you are more
```

```
                interested in only the actual function time, not counting
                the time spent in other functions that it may call, then
                disable the graph-time option.

  record-cmd - The task names when traced are recorded into a small buffer
                during task switches. For tracers this is automatic, and by
                default, event tracing will do the same. But if you do not
                care about the name of the task, disable "record-cmd".

    record-cmd:

            bash-5288  [003] 10196.848377: irq_handler_entry: irq=21 name=eth0

    norecord-cmd:

            <...>-5288  [003] 10196.848377: irq_handler_entry: irq=21 name=eth0

  overwrite - By default, when the ring buffer fills up on a CPU, it will start
              overwriting the older data to make room for the newer data.
              If you prefer a producer/consumer approach where the writer must
wait
              for the reader, then disable "overwrite".

  disable_on_free - By writing any value into the file free_buffer, will cause the
                    ring buffer to shrink to a minimum, and all allocated pages
                    will be freed. If this option is set, writing into the ring
buffer
                    will be disabled as well. The ring buffer still maintains a
few
                    pages when set to its minimum, but it may not make sense to
keep
                    writing to it.

Some options only appear when a tracer is set in current_tracer:

 [function]
   func_stack_trace - This is similar to the stacktrace option for events. When
                      enabled, each function that is traced will have its stack
dump
                      as well.

                 CAUTION: Do not enable this for all functions, it may cause the
system
                 to live lock. Think about it, every kernel function called is not
only
                 being traced, but having its stack traced as well.

                 Only use it when filtering a few functions. See set_ftrace_filter
below.

 [function_graph]
   funcgraph-overrun - Function graph tracing only traces a finite depth within a
                       function. If it exceeds this depth, then it is considered
an
                       overrun. If you are interested to see if any overruns
happened,
                       enable "funcgraph-overrun".

   funcgraph-cpu - By default the output shows the CPU number for each trace entry.
                   To suppress this, disable "funcgraph-cpu".
```

```
   funcgraph-overhead - By default, if a function took over a certain amount of
time,
                        a character is displayed by the duration.
                        ('!' - greater than 100us, '+' greater than 1us)
                        To suppress this, disable "funcgraph-overhead".

   funcgraph-proc - By default (to save room), the task and pid are not shown in
output
                    of the trace. To display these, enable "funcgraph-proc".

   funcgraph-duration - By default, the time each function took is displayed in the
                        output. To suppress this, disable "funcgraph-duration".

   funcgraph-abstime - By default (to save room), the timestamp is not displayed.
                       To display the absolute timestamp, enable "funcgraph-
abstime".

   funcgraph-irqs - By default, when an interrupt is detected, it is annotated with
                    "==========>" on entry, and "<==========" on exit of the
                    interrupt. To suppress this, disable "funcgraph-irqs".

  [wakeup, wakeup_rt, irqsoff, preemptoff, preemptirqsoff]

  display-graph: By default, the latency tracers (irqsoff, preempoff,
preempirqsoff,
                 wakeup, and wakeup_rt) will show a function trace where the
latency
                 was detected. By enabling "display-graph" and having function
graph
                 tracer configured in, the latency tracers will use the function
                 graph tracer instead. Note, the function graph tracer produces
                 much more overhead than the function tracer, which will make the
latencies
                 even larger. Requires that ftrace_enabled is set (see next
section).

ftrace_enabled
--------------

The following tracers (listed below) give different output
depending on whether or not the sysctl ftrace_enabled is set. To
set ftrace_enabled, one can either use the sysctl function or
set it via the proc file system interface.

  sysctl kernel.ftrace_enabled=1

 or

  echo 1 > /proc/sys/kernel/ftrace_enabled

To disable ftrace_enabled simply replace the '1' with '0' in the
above commands.

When ftrace_enabled is set the latency tracers will also record the
functions that are within the trace. The following descriptions of the
tracers will also show an example with ftrace enabled.


irqsoff
```

```
-------

When interrupts are disabled, the CPU can not react to any other
external event (besides NMIs and SMIs). This prevents the timer
interrupt from triggering or the mouse interrupt from letting
the kernel know of a new mouse event. The result is a latency
with the reaction time.

The irqsoff tracer tracks the time for which interrupts are
disabled. When a new maximum latency is hit, the tracer saves
the trace leading up to that latency point so that every time a
new maximum is reached, the old saved trace is discarded and the
new trace is saved.

To reset the maximum, echo 0 into tracing_max_latency. Here is
an example:

 # echo irqsoff > current_tracer
 # echo 0 > tracing_max_latency
 # ls -ltr
 [...]
 # cat trace
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.2.16-rt27-mrg-test
# --------------------------------------------------------------------
# latency: 88 us, #4/4, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: sirq-timer/3-48 (uid:0 nice:0 policy:1 rt_prio:49)
#    -----------------
#  => started at: save_args
#  => ended at:   run_ksoftirqd
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /_--=> lock-depth
#                |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
   <...>-3571    3d....    0us+: trace_hardirqs_off_thunk <-save_args
sirq-tim-48      3d....   87us : schedule <-run_ksoftirqd
sirq-tim-48      3d....   89us : trace_hardirqs_on <-run_ksoftirqd
sirq-tim-48      3d....   90us : <stack trace>
 => schedule
 => run_ksoftirqd
 => kthread
 => kernel_thread_helper

Here we see that that we had a latency of 88 microsecs.
The trace_hardirqs_off_thunk is a helper routine in the assembly
code of save_args that disabled interrupts. The difference
between the 88 and the displayed timestamp 90us occurred because the
clock was incremented between the time of recording the max latency
and the time of recording the function that had that latency.

At the end of the trace, a stack dump is given to help find the
```

```
full call graph of the location that had the irqsoff latency.

Note the above example had ftrace_enabled not set. If we set the
ftrace_enabled, we get a much larger output:

# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.2.16-rt27-mrg-test
# --------------------------------------------------------------------
# latency: 300 us, #301/301, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: sirq-sched/3-54 (uid:0 nice:0 policy:1 rt_prio:49)
#    -----------------
#  => started at: save_args
#  => ended at:   run_ksoftirqd
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /_--=> lock-depth
#                |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
   <...>-3593    3d....    1us : trace_hardirqs_off_thunk <-save_args
   <...>-3593    3d....    2us : smp_apic_timer_interrupt <-apic_timer_interrupt
   <...>-3593    3d....    3us : ack_APIC_irq <-smp_apic_timer_interrupt
   <...>-3593    3d....    4us : apic_write <-ack_APIC_irq
   <...>-3593    3d....    5us : native_apic_mem_write <-apic_write
   <...>-3593    3d....    6us : exit_idle <-smp_apic_timer_interrupt
   <...>-3593    3d....    7us : irq_enter <-smp_apic_timer_interrupt
   <...>-3593    3d....    8us : rcu_irq_enter <-irq_enter
   <...>-3593    3d....    9us : idle_cpu <-irq_enter
   <...>-3593    3d.h..   10us : hrtimer_interrupt <-smp_apic_timer_interrupt
   <...>-3593    3d.h..   11us+: ktime_get <-hrtimer_interrupt
   <...>-3593    3d.h..   13us : timekeeping_get_ns <-ktime_get
   <...>-3593    3d.h..   13us : ktime_set <-ktime_get
   <...>-3593    3d.h..   15us : _raw_spin_lock <-hrtimer_interrupt
[...]
   <...>-3593    3d..2.  286us : account_group_exec_runtime <-update_curr
   <...>-3593    3d..2.  287us : check_spread.clone.63 <-put_prev_task_fair
   <...>-3593    3d..2.  288us : __enqueue_entity <-put_prev_task_fair
   <...>-3593    3d..2.  288us : pick_next_task <-__schedule
   <...>-3593    3d..2.  289us : pick_next_task_rt <-pick_next_task
   <...>-3593    3d..2.  290us : sched_find_first_bit <-pick_next_task_rt
   <...>-3593    3d..2.  291us : sched_info_queued <-__schedule
   <...>-3593    3d..2.  292us : atomic_inc <-__schedule
   <...>-3593    3d..2.  293us : enter_lazy_tlb.clone.15 <-__schedule
   <...>-3593    3d..2.  294us : native_load_tls <-__switch_to
   <...>-3593    3d..2.  295us+: __unlazy_fpu <-__switch_to
sirq-sch-54      3d..2.  296us : finish_task_switch <-__schedule
sirq-sch-54      3d..2.  297us : _raw_spin_unlock <-finish_task_switch
sirq-sch-54      3d..1.  298us : test_ti_thread_flag.clone.2 <-_raw_spin_unlock
sirq-sch-54      3d....  299us : post_schedule <-__schedule
sirq-sch-54      3d....  300us : schedule <-run_ksoftirqd
sirq-sch-54      3d....  301us : trace_hardirqs_on <-run_ksoftirqd
sirq-sch-54      3d....  302us : <stack trace>
 => schedule
```

```
 => run_ksoftirqd
 => kthread
 => kernel_thread_helper


Here we traced a 300 microsecond latency. But we also see all the
functions that were called during that time. Note that by
enabling function tracing, we incur an added overhead. This
overhead causes the latency times to be greatly exaggerated.
The previous largest time was 88us has grown to 300us for the
same latency. But nevertheless, this trace has provided some very
helpful debugging information.

If the option "display-graph" is enabled, the following output would appear.

# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 3.2.16-test-rt27
# --------------------------------------------------------------------
# latency: 122 us, #259/259, CPU#3 | (Mreempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: swapper/3-0 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: __schedule
#  => ended at:   return_to_handler
#
#
#                                      _-----=> irqs-off
#                                     / _----=> need-resched
#                                    | / _---=> hardirq/softirq
#                                    || / _--=> preempt-depth
#                                    ||| /
#     TIME        CPU  TASK/PID      ||||  DURATION                  FUNCTION
CALLS
#      |          |     |    |       ||||   |   |                     |   |   |
|
  145.917252 |   3)   ksoftir-20   |  d..1.  0.000 us    |  _raw_spin_lock_irq();
  145.917252 |   3)   ksoftir-20   |  d..1.  0.084 us    |  add_preempt_count();
  145.917253 |   3)   ksoftir-20   |  d..2.  0.097 us    |  do_raw_spin_lock();
  145.917253 |   3)   ksoftir-20   |  d..2.              |  signal_pending_state()
{
  145.917254 |   3)   ksoftir-20   |  d..2.  0.078 us    |
test_ti_thread_flag();
  145.917254 |   3)   ksoftir-20   |  d..2.  0.644 us    |  }
  145.917254 |   3)   ksoftir-20   |  d..2.              |  deactivate_task() {
  145.917255 |   3)   ksoftir-20   |  d..2.              |    dequeue_task() {
  145.917255 |   3)   ksoftir-20   |  d..2.  0.142 us    |      update_rq_clock();
  145.917256 |   3)   ksoftir-20   |  d..2.              |      dequeue_task_rt()
{
  145.917256 |   3)   ksoftir-20   |  d..2.              |        update_curr_rt()
{
  145.917256 |   3)   ksoftir-20   |  d..2.  0.085 us    |
account_group_exec_runtime();
  145.917257 |   3)   ksoftir-20   |  d..2.              |
cpuacct_charge() {
  145.917257 |   3)   ksoftir-20   |  d..2.  0.069 us    |
__rcu_read_lock();
  145.917258 |   3)   ksoftir-20   |  d..2.  0.066 us    |
__rcu_read_unlock();
  145.917258 |   3)   ksoftir-20   |  d..2.  1.140 us    |          }
```

```
[...]

  145.917371 |   3)   ksoftir-20   |  d..2.  0.087 us    |  atomic_inc();
  145.917372 |   3)   ksoftir-20   |  d..2.  0.072 us    |  native_load_tls();
 ------------------------------------------
 3)   ksoftir-20   =>    <idle>0
 ------------------------------------------

  145.917373 |   3)    <idle>0    |  d..2.              |  finish_task_switch() {
  145.917373 |   3)    <idle>0    |  d..2.              |
_raw_spin_unlock_irq() {
  145.917373 |   3)    <idle>0    |  d..2.  0.000 us    |  _raw_spin_unlock_irq();
  145.917374 |   3)    <idle>0    |  d..2.  0.000 us    |  trace_hardirqs_on();
  <idle>0       3d..2.  158us : <stack trace>
 => trace_hardirqs_on
 => _raw_spin_unlock_irq
 => return_to_handler
 => __schedule
 => return_to_handler
 => schedule
 => schedule_preempt_disabled
 => cpu_idle
 => start_secondary


preemptoff
----------

When preemption is disabled, we may be able to receive
interrupts but the task cannot be preempted and a higher
priority task must wait for preemption to be enabled again
before it can preempt a lower priority task.

The preemptoff tracer traces the places that disable preemption.
Like the irqsoff tracer, it records the maximum latency for
which preemption was disabled. The control of preemptoff tracer
is much like the irqsoff tracer.

 # echo preemptoff > current_tracer
 # echo 0 > tracing_max_latency
 # ls -ltr
 [...]
 # cat trace
# tracer: preemptoff
#
# preemptoff latency trace v1.1.5 on 3.2.16-rt27-mrg-test
# --------------------------------------------------------------------
# latency: 28 us, #4/4, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: irqbalance-1460 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: smp_apic_timer_interrupt
#  => ended at:   smp_apic_timer_interrupt
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
```

```
#                ||| / _--=> preempt-depth
#                |||| /_--=> lock-depth
#                |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
irqbalan-1460    0d.h..    1us+: irq_enter <-smp_apic_timer_interrupt
irqbalan-1460    0dN.1.   28us : irq_exit <-smp_apic_timer_interrupt
irqbalan-1460    0dN.1.   29us : trace_preempt_on <-smp_apic_timer_interrupt
irqbalan-1460    0dN.1.   29us : <stack trace>
 => sub_preempt_count
 => irq_exit
 => smp_apic_timer_interrupt
 => apic_timer_interrupt
 => show_stat
 => seq_read
 => proc_reg_read
 => vfs_read

This has some more changes. Preemption was disabled when an
interrupt came in (notice the 'h'), and was enabled when returning
from the softirq. The 'N' flag tells us that the NEED_RESCHED flag
of the task has been set. We also see that interrupts
have been disabled when entering the preempt off section and
leaving it (the 'd'). We do not know if interrupts were enabled
in the mean time.

# tracer: preemptoff
#
# preemptoff latency trace v1.1.5 on 3.2.16-rt27-mrg-test
# --------------------------------------------------------------------
# latency: 148 us, #167/167, CPU#2 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: bash-2040 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: default_wake_function
#  => ended at:   default_wake_function
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /_--=> lock-depth
#                |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
    bash-2040    2...1.    0us+: try_to_wake_up <-default_wake_function
    bash-2040    2d..1.    2us : _raw_spin_lock <-task_rq_lock
    bash-2040    2d..2.    3us : do_raw_spin_lock <-_raw_spin_lock
    bash-2040    2d..2.    3us : update_rq_clock <-try_to_wake_up
    bash-2040    2d..2.    5us : task_waking_fair <-try_to_wake_up
    bash-2040    2d..2.    6us : cfs_rq_of <-task_waking_fair
    bash-2040    2d..2.    6us : select_task_rq <-try_to_wake_up
    bash-2040    2d..2.    7us : select_task_rq_fair <-select_task_rq
    bash-2040    2d..2.    8us : _raw_spin_unlock <-select_task_rq_fair
    bash-2040    2d..1.    9us : test_ti_thread_flag.clone.2 <-_raw_spin_unlock
    bash-2040    2d..1.   10us : _raw_spin_lock <-select_task_rq_fair
[...]
    bash-2040    2d..2.   53us : resched_task <-check_preempt_curr_idle
```

```
    bash-2040    2d..2.   53us : test_ti_thread_flag <-resched_task
    bash-2040    2d..2.   54us : set_tsk_need_resched <-resched_task
    bash-2040    2d..2.   55us : _raw_spin_unlock_irqrestore <-try_to_wake_up
    bash-2040    2d..2.   57us : smp_apic_timer_interrupt <-apic_timer_interrupt
    bash-2040    2d..2.   57us : ack_APIC_irq <-smp_apic_timer_interrupt
    bash-2040    2d..2.   58us : apic_write <-ack_APIC_irq
    bash-2040    2d..2.   59us : native_apic_mem_write <-apic_write
    bash-2040    2d..2.   59us : exit_idle <-smp_apic_timer_interrupt
    bash-2040    2d..2.   60us : irq_enter <-smp_apic_timer_interrupt
    bash-2040    2d..2.   61us : rcu_irq_enter <-irq_enter
    bash-2040    2d..2.   61us : idle_cpu <-irq_enter
    bash-2040    2d.h2.   62us : hrtimer_interrupt <-smp_apic_timer_interrupt
[...]
    bash-2040    2dNh2.  142us : apic_write <-lapic_next_event
    bash-2040    2dNh2.  143us : native_apic_mem_write <-apic_write
    bash-2040    2dNh2.  143us : irq_exit <-smp_apic_timer_interrupt
    bash-2040    2dN.3.  144us : do_softirq <-irq_exit
    bash-2040    2dN.3.  145us : __do_softirq <-call_softirq
    bash-2040    2dN.3.  145us : trigger_softirqs <-__do_softirq
    bash-2040    2dN.3.  146us : wakeup_softirqd <-trigger_softirqs
    bash-2040    2dN.3.  146us : rcu_irq_exit <-irq_exit
    bash-2040    2dN.3.  147us : idle_cpu <-irq_exit
    bash-2040    2.N.1.  148us : try_to_wake_up <-default_wake_function
    bash-2040    2.N.1.  149us : trace_preempt_on <-default_wake_function
    bash-2040    2.N.1.  150us : <stack trace>
 => sub_preempt_count
 => try_to_wake_up
 => default_wake_function
 => autoremove_wake_function
 => __wake_up_common
 => __wake_up_sync_key
 => __wake_up_sync
 => pipe_release

The above is an example of the preemptoff trace with
ftrace_enabled set. Here we see that interrupts were enabled just
before preemption was enabled. Also, interrupts were enabled
at the _raw_spin_unlock_irqrestore() call, and at that moment
the timer interrupt (apic_timer_interrupt) came in. But because
no function was traced between those two events, the 'd' flag
was never shown to be off there.

The irq_enter code lets us know that we entered an interrupt 'h'.
Before that, the functions being traced still show that it is not
in an interrupt, but we can see from the functions themselves that
this is not the case.


preemptirqsoff
--------------

Knowing the locations that have interrupts disabled or
preemption disabled for the longest times is helpful. But
sometimes we would like to know when either preemption and/or
interrupts are disabled.

Consider the following code:

    local_irq_disable();
    call_function_with_irqs_off();
```

```
    preempt_disable();
    call_function_with_irqs_and_preemption_off();
    local_irq_enable();
    call_function_with_preemption_off();
    preempt_enable();

The irqsoff tracer will record the total length of
call_function_with_irqs_off() and
call_function_with_irqs_and_preemption_off().

The preemptoff tracer will record the total length of
call_function_with_irqs_and_preemption_off() and
call_function_with_preemption_off().

But neither will trace the time that interrupts and/or
preemption is disabled. This total time is the time that we can
not schedule. To record this time, use the preemptirqsoff
tracer.

Again, using this trace is much like the irqsoff and preemptoff
tracers.

 # echo preemptirqsoff > current_tracer
 # echo 0 > tracing_max_latency
 # ls -ltr
 [...]
 # cat trace
# tracer: preemptirqsoff
#
# preemptirqsoff latency trace v1.1.5 on 3.2.16-rt27-mrg-test
# --------------------------------------------------------------------
# latency: 52 us, #4/4, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: hackbench-11587 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: rt_spin_lock_slowunlock
#  => ended at:   rt_spin_lock_slowunlock
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /_--=> lock-depth
#                |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
hackbenc-11587   0d....    0us+: _raw_spin_lock_irqsave <-rt_spin_lock_slowunlock
hackbenc-11587   0.N.1.   52us : _raw_spin_unlock_irqrestore <-
rt_spin_lock_slowunlock
hackbenc-11587   0.N.1.   53us : trace_preempt_on <-rt_spin_lock_slowunlock
hackbenc-11587   0.N.1.   54us : <stack trace>
 => sub_preempt_count
 => _raw_spin_unlock_irqrestore
 => rt_spin_lock_slowunlock
 => rt_spin_lock_fastunlock.clone.13
 => rt_spin_unlock
 => slab_irq_enable
 => kfree
```

```
 => skb_release_data

Interrupts and preemption was disabled at the _raw_spin_lock_irqsave.
Although the interrupts are shown disabled and the preemption was not,
is just the placement of where the recording takes place (it happens
after interrupts were disabled, and before the preemption was disabled).
Both interrupts and preemption is re-enabled at the rt_spin_lock_slowunlock.
This time due to the placement of the disabling, the interrupts are
shown enabled while preemption is still disabled.

Here is a trace with ftrace_enabled set:

# tracer: preemptirqsoff
#
# preemptirqsoff latency trace v1.1.5 on 3.2.16-rt27-mrg-test
# --------------------------------------------------------------------
# latency: 350 us, #457/457, CPU#2 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: hackbench-4755 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: rt_spin_lock_slowunlock
#  => ended at:   rt_spin_lock_slowunlock
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /_--=> lock-depth
#                |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
hackbenc-4755    2d....    1us : _raw_spin_lock_irqsave <-rt_spin_lock_slowunlock
hackbenc-4755    2d..1.    2us : wakeup_next_waiter <-rt_spin_lock_slowunlock
hackbenc-4755    2d..1.    3us : rt_mutex_top_waiter <-wakeup_next_waiter
hackbenc-4755    2d..1.    3us : _raw_spin_lock <-wakeup_next_waiter
hackbenc-4755    2d..2.    4us : do_raw_spin_lock <-_raw_spin_lock
hackbenc-4755    2d..2.    5us : _raw_spin_unlock <-wakeup_next_waiter
hackbenc-4755    2d..1.    6us : test_ti_thread_flag.clone.2 <-_raw_spin_unlock
hackbenc-4755    2d..1.    6us : wake_up_process_mutex <-wakeup_next_waiter
hackbenc-4755    2d..1.    7us : try_to_wake_up <-wake_up_process_mutex
hackbenc-4755    2d..2.    8us : task_rq_lock <-try_to_wake_up
hackbenc-4755    2d..2.    8us : __raw_local_irq_save <-task_rq_lock
hackbenc-4755    2d..2.    9us : __raw_local_save_flags <-__raw_local_irq_save
[...]
hackbenc-4755    2d..3.   20us : wakeup_preempt_entity <-check_preempt_wakeup
hackbenc-4755    2d..3.   21us : _raw_spin_unlock_irqrestore <-try_to_wake_up
hackbenc-4755    2d..2.   21us : test_ti_thread_flag.clone.2 <-
_raw_spin_unlock_irqrestore
hackbenc-4755    2d..1.   22us : test_ti_thread_flag <-try_to_wake_up
hackbenc-4755    2d..1.   23us+: _raw_spin_unlock_irqrestore <-
rt_spin_lock_slowunlock
hackbenc-4755    2d..1.   25us : do_IRQ <-ret_from_intr
hackbenc-4755    2d..1.   26us : exit_idle <-do_IRQ
hackbenc-4755    2d..1.   26us : irq_enter <-do_IRQ
hackbenc-4755    2d..1.   27us : rcu_irq_enter <-irq_enter
hackbenc-4755    2d..1.   28us : idle_cpu <-irq_enter
hackbenc-4755    2d.h1.   29us : handle_irq <-do_IRQ
hackbenc-4755    2d.h1.   30us : irq_to_desc <-handle_irq
```

```
[...]
hackbenc-4755    2dNh1.  271us : timekeeping_get_ns <-ktime_get
hackbenc-4755    2dNh1.  272us : clockevents_program_event <-
tick_dev_program_event
hackbenc-4755    2dNh1.  272us : lapic_next_event <-clockevents_program_event
hackbenc-4755    2dNh1.  273us : apic_write <-lapic_next_event
hackbenc-4755    2dNh1.  274us : native_apic_mem_write <-apic_write
hackbenc-4755    2dNh1.  274us : irq_exit <-smp_apic_timer_interrupt
hackbenc-4755    2dN.2.  275us : do_softirq <-irq_exit
hackbenc-4755    2dN.2.  276us : __do_softirq <-call_softirq
hackbenc-4755    2dN.2.  277us : trigger_softirqs <-__do_softirq
hackbenc-4755    2dN.2.  277us : wakeup_softirqd <-trigger_softirqs
hackbenc-4755    2dN.2.  278us : rcu_irq_exit <-irq_exit
hackbenc-4755    2dN.2.  279us+: idle_cpu <-irq_exit
[...]
hackbenc-4755    2dNh1.  343us : irq_exit <-do_IRQ
hackbenc-4755    2dN.2.  344us : do_softirq <-irq_exit
hackbenc-4755    2dN.2.  345us : __do_softirq <-call_softirq
hackbenc-4755    2dN.2.  346us : trigger_softirqs <-__do_softirq
hackbenc-4755    2dN.2.  346us : wakeup_softirqd <-trigger_softirqs
hackbenc-4755    2dN.2.  348us : rcu_irq_exit <-irq_exit
hackbenc-4755    2dN.2.  349us : idle_cpu <-irq_exit
hackbenc-4755    2.N.1.  350us : _raw_spin_unlock_irqrestore <-
rt_spin_lock_slowunlock
hackbenc-4755    2.N.1.  351us : trace_preempt_on <-rt_spin_lock_slowunlock
hackbenc-4755    2.N.1.  352us : <stack trace>
 => sub_preempt_count
 => _raw_spin_unlock_irqrestore
 => rt_spin_lock_slowunlock
 => rt_spin_lock_fastunlock.clone.13
 => rt_spin_unlock
 => slab_irq_enable
 => kmem_cache_alloc_node
 => __alloc_skb


This is a very interesting trace. It started again with the irq disabling
of _raw_spin_lock_irqsave which also disabled preemption later.
But we can also see here that when it enabled interrupts before
disabling preemption, the time interrupt triggered. As the interrupt exited,
it enabled softirqs. Finally when the interrupt returned, the
_raw_spin_unlock_irqrestore was able to disable preemption. If we
did not have the function tracer running, we would not have noticed
that an interrupt arrived. (But we can if we enabled events,
see events.txt for more info.)


wakeup and wakeup_rt
--------------------

In a Real-Time environment it is very important to know the
wakeup time it takes for the highest priority task that is woken
up to the time that it executes. This is also known as "schedule
latency".

Real-Time environments are interested in the worst case latency.
That is the longest latency it takes for something to happen,
and not the average. We can have a very fast scheduler that may
only have a large latency once in a while, but that would not
work well with Real-Time tasks. The wakeup_rt tracer was designed
```

```
to record the worst case wakeups of RT tasks. Non-RT tasks are
not recorded because the tracer only records one worst case and
tracing non-RT tasks that are unpredictable will overwrite the
worst case latency of RT tasks. If you are still interested in non-RT
tasks, then use the wakeup tracer.

Since the wakeup_rt tracer only deals with RT tasks, we will run this
slightly differently than we did with the previous tracers.
Instead of performing an 'ls', we will run 'sleep 1' under
'chrt' which changes the priority of the task.

 # echo wakeup_rt > current_tracer
 # echo 0 > tracing_max_latency
 # chrt -f 90 sleep 1
 # cat trace
# tracer: wakeup_rt
#
# wakeup_rt latency trace v1.1.5 on 3.2.16-test-rt27
# --------------------------------------------------------------------
# latency: 5 us, #4/4, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: sleep-2903 (uid:0 nice:0 policy:1 rt_prio:90)
#    -----------------
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /_--=> migrate-disable
#                ||||| /     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      |||||  \   |   /
  <idle>0       3d.h4.    0us :      0:120:R   + [003]  2903: 49:R sleep
  <idle>0       3d.h4.    0us+: ttwu_do_activate.constprop.175 <try_to_wake_up
  <idle>0       3d..3.    6us : __schedule <schedule
  <idle>0       3d..3.    6us :      0:120:R ==> [003]  2903: 9:R sleep

Running this on an idle system, we see that it took 5
microseconds to perform the task switch.

The header shows the task PID of 2903 that was recorded.
The rt_prio is 90 which is the user space priority. The prio shown
in the wake up and schedule events is 9 which is the kernel version
of that priority.  The policy is 1 for SCHED_FIFO and 2 for SCHED_RR.

Remember that the KERNEL-PRIO is the inverse of the actual
priority with zero (0) being the highest priority and the nice
values starting at 100 (nice -20). Below is a quick chart to map
the kernel priority to user land priorities.

   Kernel Space                     User Space
 ===============================================================
   0(high) to  98(low)     user RT priority 99(high) to 1(low)
                            with SCHED_RR or SCHED_FIFO
 ---------------------------------------------------------------
  99                        sched_priority is not used in scheduling
                            decisions(it must be specified as 0)
 ---------------------------------------------------------------
 100(high) to 139(low)     user nice -20(high) to 19(low)
```

```
 -----------------------------------------------------------------

The task states, like in the final event:

 R - running : wants to run, may not actually be running
 S - sleep   : process is waiting to be woken up (handles signals)
 D - disk sleep (uninterruptible sleep) : process must be woken up
                                         (ignores signals)
 T - stopped : process suspended
 t - traced  : process is being traced (with something like gdb)
 Z - zombie  : process waiting to be cleaned up
 X - unknown


Doing the same with chrt -r 90 and ftrace_enabled set.

# tracer: wakeup
#
# wakeup latency trace v1.1.5 on 3.2.16-test
# -----------------------------------------------------------------------
# latency: 95 us, #179/179, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: -6 (uid:0 nice:0 policy:1 rt_prio:99)
#    -----------------
#
#                   _------=> CPU#
#                  / _-----=> irqs-off
#                 | / _----=> need-resched
#                 || / _---=> hardirq/softirq
#                 ||| / _--=> preempt-depth
#                 |||| /_--=> lock-depth
#                 |||||/     delay
#  cmd     pid   |||||| time  |   caller
#     \   /      ||||||   \   |   /
   sleep-2461    1d..3.    1us+:   2461:120:R   + [001]     6:  0:R migration/1
   sleep-2461    1d..3.    5us : wake_up_process <-sched_exec
   sleep-2461    1d..2.    6us : test_ti_thread_flag <-
rcu_read_unlock_sched_notrace
   sleep-2461    1d..2.    6us : check_preempt_curr <-try_to_wake_up
   sleep-2461    1d..2.    7us : check_preempt_wakeup <-check_preempt_curr
   sleep-2461    1d..2.    7us : resched_task <-check_preempt_wakeup
   sleep-2461    1d..2.    8us : test_ti_thread_flag <-resched_task
   sleep-2461    1d..2.    8us : set_tsk_need_resched <-resched_task
   sleep-2461    1dN.2.    9us : task_woken_rt <-try_to_wake_up
   sleep-2461    1dN.2.    9us : test_ti_thread_flag <-task_woken_rt
   sleep-2461    1dN.2.   10us : _raw_spin_unlock_irqrestore <-try_to_wake_up
   sleep-2461    1dN.2.   11us : smp_apic_timer_interrupt <-apic_timer_interrupt
   sleep-2461    1dN.2.   11us : ack_APIC_irq <-smp_apic_timer_interrupt
   sleep-2461    1dN.2.   12us : apic_write <-ack_APIC_irq
   sleep-2461    1dN.2.   12us : native_apic_mem_write <-apic_write
   sleep-2461    1dN.2.   13us : exit_idle <-smp_apic_timer_interrupt
   sleep-2461    1dN.2.   13us : irq_enter <-smp_apic_timer_interrupt
   sleep-2461    1dN.2.   14us : rcu_irq_enter <-irq_enter
   sleep-2461    1dN.2.   14us : idle_cpu <-irq_enter
   sleep-2461    1dNh2.   15us : hrtimer_interrupt <-smp_apic_timer_interrupt
[...]
   sleep-2461    1dNh2.   56us : native_apic_mem_write <-apic_write
   sleep-2461    1dNh2.   57us : irq_exit <-smp_apic_timer_interrupt
   sleep-2461    1dN.3.   57us : do_softirq <-irq_exit
   sleep-2461    1dN.3.   58us : __do_softirq <-call_softirq
```

```
   sleep-2461    1dN.3.   58us : __local_bh_disable <-__do_softirq
   sleep-2461    1dN.3.   58us : __raw_local_irq_save <-__local_bh_disable
   sleep-2461    1dN.3.   59us : __raw_local_save_flags <-__raw_local_irq_save
   sleep-2461    1.Ns3.   60us : run_timer_softirq <-__do_softirq
   sleep-2461    1.Ns3.   61us : hrtimer_run_pending <-run_timer_softirq
[...]
   sleep-2461    1.Ns3.   80us : rcu_bh_qs <-__do_softirq
   sleep-2461    1dNs3.   80us : _local_bh_enable <-__do_softirq
   sleep-2461    1dNs3.   80us : __raw_local_save_flags <-_local_bh_enable
   sleep-2461    1dN.3.   81us : rcu_irq_exit <-irq_exit
   sleep-2461    1dN.3.   82us : idle_cpu <-irq_exit
   sleep-2461    1.N.1.   82us : test_ti_thread_flag.clone.2 <-
_raw_spin_unlock_irqrestore
   sleep-2461    1.N.1.   83us : preempt_schedule <-_raw_spin_unlock_irqrestore
   sleep-2461    1.N...   83us : test_ti_thread_flag <-try_to_wake_up
   sleep-2461    1.N...   84us : preempt_schedule <-try_to_wake_up
   sleep-2461    1.N...   84us : __raw_local_save_flags <-preempt_schedule
   sleep-2461    1.N...   85us : schedule <-preempt_schedule
   sleep-2461    1.N.1.   85us : rcu_sched_qs <-schedule
[...]
   sleep-2461    1d..2.   92us : pick_next_task_rt <-pick_next_task
   sleep-2461    1d..2.   93us : sched_find_first_bit <-pick_next_task_rt
   sleep-2461    1d..3.   94us : schedule <-preempt_schedule
   sleep-2461    1d..3.   95us :   2461:120:R ==> [001]     6:  0:R migration/1

This time instead of tracing the wakeup of our sleep task, the trace
captured the migration task. It may have caught the sleep task, but
then the migration task took longer to wake up, and only the maximum
trace is stored. Shortly after the migration thread was worken up on
the same CPU our sleep task was running "[001]", the sleep task
need resched flag was set ("N"). After "_raw_spin_unlock_irqrestore()"
enabled interrupts, a timer interrupt triggered (also disabling
interrupts leaving the 'd' set). The irq_entry() has a hook to cause
the 'h' flag to be set to show that the event happened in interrupt
context. The timer interrupt queued the timer softirq and then started
executing that. The 's' flag shows the softirqs are disabled or is
running. Finally, the softirq returns back to the original place
the code was interrupted and the migration thread is scheduled.

function
--------

This tracer is the function tracer. Enabling the function tracer
can be done from the debug file system. Make sure the
ftrace_enabled is set; otherwise this tracer is a nop. On boot up
the ftrace_enabled sysctl is set, but the bootup scripts or a user
could have cleared it.

 # sysctl kernel.ftrace_enabled=1
 # echo function > current_tracer
 # usleep 1
 # cat trace
 # echo 0 > tracing_on
# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
          <idle>-0     [003] 15774.015440: test_ti_thread_flag <-cpu_idle
          <idle>-0     [003] 15774.015441: enter_idle <-cpu_idle
          <idle>-0     [003] 15774.015442: mwait_idle <-cpu_idle
```

```
          <idle>-0     [003] 15774.015442: need_resched <-mwait_idle
          <idle>-0     [003] 15774.015443: test_ti_thread_flag <-need_resched
          <idle>-0     [003] 15774.015444: trace_power_start.clone.5 <-mwait_idle
          <idle>-0     [003] 15774.015445: need_resched <-mwait_idle
          <idle>-0     [003] 15774.015446: test_ti_thread_flag <-need_resched
          <idle>-0     [003] 15774.015447: __exit_idle <-cpu_idle
          <idle>-0     [003] 15774.015448: test_ti_thread_flag <-cpu_idle
          <idle>-0     [003] 15774.015449: enter_idle <-cpu_idle
          <idle>-0     [003] 15774.015450: mwait_idle <-cpu_idle
          <idle>-0     [003] 15774.015450: need_resched <-mwait_idle
          <idle>-0     [003] 15774.015451: test_ti_thread_flag <-need_resched
          <idle>-0     [003] 15774.015452: trace_power_start.clone.5 <-mwait_idle
          <idle>-0     [003] 15774.015453: need_resched <-mwait_idle
          <idle>-0     [003] 15774.015454: test_ti_thread_flag <-need_resched
          <idle>-0     [003] 15774.015455: __exit_idle <-cpu_idle
[...]


Note: function tracer uses ring buffers to store the above
entries. The newest data may overwrite the oldest data (unless the
overwrite option is off) Sometimes using echo to stop the trace is not
sufficient because the tracing could have overwritten the data that
you wanted to record. For this reason, it is sometimes better to disable
tracing directly from a program. This allows you to stop the
tracing at the point that you hit the part that you are
interested in. To disable the tracing directly from a C program,
something like following code snippet can be used:

int trace_fd;
[...]
int main(int argc, char *argv[]) {
 [...]
 trace_fd = open(tracing_file("tracing_on"), O_WRONLY);
 [...]
 if (condition_hit()) {
  write(trace_fd, "0", 1);
 }
 [...]
}


Single thread tracing
---------------------

By writing into set_ftrace_pid you can trace a
single thread. For example:

# cat set_ftrace_pid
no pid
# echo 3111 > set_ftrace_pid
# cat set_ftrace_pid
3111
# echo function > current_tracer
# cat trace | head
 # tracer: function
 #
 #            TASK-PID    CPU#    TIMESTAMP  FUNCTION
 #               | |       |          |         |
     yum-updatesd-3111  [003]  1637.254676: finish_task_switch <-thread_return
     yum-updatesd-3111  [003]  1637.254681: hrtimer_cancel <-
```

```
schedule_hrtimeout_range
     yum-updatesd-3111  [003]  1637.254682: hrtimer_try_to_cancel <-
hrtimer_cancel
     yum-updatesd-3111  [003]  1637.254683: lock_hrtimer_base <-
hrtimer_try_to_cancel
     yum-updatesd-3111  [003]  1637.254685: fget_light <-do_sys_poll
     yum-updatesd-3111  [003]  1637.254686: pipe_poll <-do_sys_poll
# echo -1 > set_ftrace_pid
# cat trace |head
 # tracer: function
 #
 #           TASK-PID    CPU#    TIMESTAMP  FUNCTION
 #              | |       |          |         |
 ##### CPU 3 buffer started ####
     yum-updatesd-3111  [003]  1701.957688: free_poll_entry <-poll_freewait
     yum-updatesd-3111  [003]  1701.957689: remove_wait_queue <-free_poll_entry
     yum-updatesd-3111  [003]  1701.957691: fput <-free_poll_entry
     yum-updatesd-3111  [003]  1701.957692: audit_syscall_exit <-sysret_audit
     yum-updatesd-3111  [003]  1701.957693: path_put <-audit_syscall_exit

If you want to trace a function when executing, you could use a
simple shell script:

   --------
#!/bin/bash
echo $$ > /sys/kernel/debug/tracing/set_ftrace_pid
echo function > /sys/kernel/debug/tracing/current_tracer
exec $*
something like this simple program:
   --------

Then just run the script followed by a program and its arguments.

For including this in a C program:

   --------
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define _STR(x) #x
#define STR(x) _STR(x)
#define MAX_PATH 256

const char *find_debugfs(void)
{
       static char debugfs[MAX_PATH+1];
       static int debugfs_found;
       char type[100];
       FILE *fp;

       if (debugfs_found)
               return debugfs;

       if ((fp = fopen("/proc/mounts","r")) == NULL) {
               perror("/proc/mounts");
```

```
                return NULL;
        }

        while (fscanf(fp, "%*s %"
                      STR(MAX_PATH)
                      "s %99s %*s %*d %*d\n",
                      debugfs, type) == 2) {
                if (strcmp(type, "debugfs") == 0)
                        break;
        }
        fclose(fp);

        if (strcmp(type, "debugfs") != 0) {
                fprintf(stderr, "debugfs not mounted");
                return NULL;
        }

        strcat(debugfs, "/tracing/");
        debugfs_found = 1;

        return debugfs;
}

const char *tracing_file(const char *file_name)
{
        static char trace_file[MAX_PATH+1];
        snprintf(trace_file, MAX_PATH, "%s/%s", find_debugfs(), file_name);
        return trace_file;
}

int main (int argc, char **argv)
{
         if (argc < 1)
                 exit(-1);

         if (fork() > 0) {
                 int fd, ffd;
                 char line[64];
                 int s;

                 ffd = open(tracing_file("current_tracer"), O_WRONLY);
                 if (ffd < 0)
                         exit(-1);
                 write(ffd, "nop", 3);

                 fd = open(tracing_file("set_ftrace_pid"), O_WRONLY);
                 s = sprintf(line, "%d\n", getpid());
                 write(fd, line, s);

                 write(ffd, "function", 8);

                 close(fd);
                 close(ffd);

                 execvp(argv[1], argv+1);
         }

         return 0;
}
   --------
```

```
function graph tracer
---------------------------

This tracer is similar to the function tracer except that it
probes a function on its entry and its exit. This is done by
using a dynamically allocated stack of return addresses in each
task_struct. On function entry the tracer overwrites the return
address of each function traced to set a custom probe. Thus the
original return address is stored on the stack of return address
in the task_struct.

Probing on both ends of a function leads to special features
such as:

- measure of a function's time execution
- having a reliable call stack to draw function calls graph

This tracer is useful in several situations:

- you want to find the reason of a strange kernel behavior and
  need to see what happens in detail on any areas (or specific
  ones).

- you are experiencing weird latencies but it's difficult to
  find its origin.

- you want to find quickly which path is taken by a specific
  function

- you just want to peek inside a working kernel and want to see
  what happens there.

# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |

 0)               |  sys_open() {
 0)               |    do_sys_open() {
 0)               |      getname() {
 0)               |        kmem_cache_alloc() {
 0)   1.382 us    |          __might_sleep();
 0)   2.478 us    |        }
 0)               |        strncpy_from_user() {
 0)               |          might_fault() {
 0)   1.389 us    |            __might_sleep();
 0)   2.553 us    |          }
 0)   3.807 us    |        }
 0)   7.876 us    |      }
 0)               |      alloc_fd() {
 0)   0.668 us    |        _spin_lock();
 0)   0.570 us    |        expand_files();
 0)   0.586 us    |        _spin_unlock();


There are several columns that can be dynamically
enabled/disabled. You can use every combination of options you
want, depending on your needs.
```

```
- The cpu number on which the function executed is default
  enabled.  It is sometimes better to only trace one cpu (see
  tracing_cpu_mask file) or you might sometimes see unordered
  function calls while cpu tracing switch.

 hide: echo nofuncgraph-cpu > trace_options
 show: echo funcgraph-cpu > trace_options

- The duration (function's time of execution) is displayed on
  the closing bracket line of a function or on the same line
  than the current function in case of a leaf one. It is default
  enabled.

 hide: echo nofuncgraph-duration > trace_options
 show: echo funcgraph-duration > trace_options

- The overhead field precedes the duration field in case of
  reached duration thresholds.

 hide: echo nofuncgraph-overhead > trace_options
 show: echo funcgraph-overhead > trace_options
 depends on: funcgraph-duration

  ie:

  0)                |    up_write() {
  0)   0.646 us    |      _spin_lock_irqsave();
  0)   0.684 us    |      _spin_unlock_irqrestore();
  0)   3.123 us    |    }
  0)   0.548 us    |    fput();
  0) + 58.628 us   |  }

  [...]

  0)                |      putname() {
  0)                |        kmem_cache_free() {
  0)   0.518 us    |          __phys_addr();
  0)   1.757 us    |        }
  0)   2.861 us    |      }
  0) ! 115.305 us  |    }
  0) ! 116.402 us  |  }

  + means that the function exceeded 10 usecs.
  ! means that the function exceeded 100 usecs.


- The task/pid field displays the thread cmdline and pid which
  executed the function. It is default disabled.

 hide: echo nofuncgraph-proc > trace_options
 show: echo funcgraph-proc > trace_options

  ie:
  # tracer: function_graph
  #
  # CPU  TASK/PID        DURATION                  FUNCTION CALLS
  # |    |    |           |   |                     |   |   |   |
  0)    sh-4802     |               |                  d_free() {
  0)    sh-4802     |               |                    call_rcu() {
```

```
  0)    sh-4802     |               |                       __call_rcu() {
  0)    sh-4802     |   0.616 us    |
rcu_process_gp_end();
  0)    sh-4802     |   0.586 us    |
check_for_new_grace_period();
  0)    sh-4802     |   2.899 us    |                       }
  0)    sh-4802     |   4.040 us    |                     }
  0)    sh-4802     |   5.151 us    |                   }
  0)    sh-4802     | + 49.370 us   |                 }


- The absolute time field is an absolute timestamp given by the
  system clock since it started. A snapshot of this time is
  given on each entry/exit of functions

 hide: echo nofuncgraph-abstime > trace_options
 show: echo funcgraph-abstime > trace_options

  ie:

  #
  #      TIME       CPU  DURATION                  FUNCTION CALLS
  #       |         |     |   |                     |   |   |   |
  360.774522 |   1)   0.541 us    |                                          }
  360.774522 |   1)   4.663 us    |                                        }
  360.774523 |   1)   0.541 us    |
__wake_up_bit();
  360.774524 |   1)   6.796 us    |                                      }
  360.774524 |   1)   7.952 us    |                                    }
  360.774525 |   1)   9.063 us    |                                  }
  360.774525 |   1)   0.615 us    |
journal_mark_dirty();
  360.774527 |   1)   0.578 us    |                                  __brelse();
  360.774528 |   1)               |
reiserfs_prepare_for_journal() {
  360.774528 |   1)               |
unlock_buffer() {
  360.774529 |   1)               |
wake_up_bit() {
  360.774529 |   1)               |
bit_waitqueue() {
  360.774530 |   1)   0.594 us    |
__phys_addr();


You can put some comments on specific functions by using
trace_printk() For example, if you want to put a comment inside
the __might_sleep() function, you just have to call trace_printk()
inside __might_sleep()

trace_printk("I'm a comment!\n")

will produce:

 1)               |             __might_sleep() {
 1)               |                /* I'm a comment! */
 1)   1.449 us    |             }


You might find other useful features for this tracer in the
```

```
following "dynamic ftrace" section such as tracing only specific
functions or tasks.

dynamic ftrace
--------------

how it works
------------

(skip this section if you do not care how dynamic ftrace
 is implemented)

If CONFIG_DYNAMIC_FTRACE is set, the system will run with
virtually no overhead when function tracing is disabled. The way
this works is the mcount function call (placed at the start of
every kernel function, produced by the -pg switch in gcc),
starts of pointing to a simple return. (Enabling FTRACE will
include the -pg switch in the compiling of the kernel.)

At compile time every C file object is run through the
the recordmcount program (located in the scripts directory). This
program will parse the ELF data within the object file and
create a new .text section that will hold all the mcount locations.

A new section called "__mcount_loc" is created that holds
references to all the mcount call sites in the .text section.
The final linker will add all these references into a single table

On boot up, before SMP is initialized, the dynamic ftrace code
scans this table and updates all the locations into nops. It
also records the locations, which are added to the
available_filter_functions list.  Modules are processed as they
are loaded and before they are executed.  When a module is
unloaded, it also removes its functions from the ftrace function
list. This is automatic in the module unload code, and the
module author does not need to worry about it.

When tracing is enabled, stop_machine is called to prevent
races with the CPUS executing code being modified (which can
cause the CPU to do undesirable things), and the nops are
patched back to calls. But this time, they do not call mcount
(which is just a function stub). They now call into the ftrace
infrastructure.

One special side-effect to the recording of the functions being
traced is that we can now selectively choose which functions we
wish to trace and which ones we want the mcount calls to remain
as nops.


Picking specific functions to trace
-----------------------------------

Two files are used, one for enabling and one for disabling the
tracing of specified functions. They are:

  set_ftrace_filter

and
```

```
  set_ftrace_notrace

A list of available functions that you can add to these files is
listed in:

   available_filter_functions

 # cat available_filter_functions
put_prev_task_idle
kmem_cache_create
pick_next_task_rt
get_online_cpus
pick_next_task_fair
mutex_lock
[...]

If I am only interested in sys_nanosleep and hrtimer_interrupt:

 # echo sys_nanosleep hrtimer_interrupt > set_ftrace_filter
 # echo function > current_tracer
 # echo 1 > tracing_on
 # usleep 1
 # echo 0 > tracing_on
 # cat trace
# tracer: ftrace
#
#  TASK-PID   CPU#    TIMESTAMP  FUNCTION
#      | |      |          |         |
  <idle>-0     [001] 33979.796281: hrtimer_interrupt <-smp_apic_timer_interrupt
  <idle>-0     [000] 33979.797217: hrtimer_interrupt <-smp_apic_timer_interrupt
  <idle>-0     [000] 33979.804207: hrtimer_interrupt <-smp_apic_timer_interrupt
  usleep-2672  [002] 33979.804330: hrtimer_interrupt <-smp_apic_timer_interrupt
  usleep-2672  [002] 33979.804785: sys_nanosleep <-system_call_fastpath
  <idle>-0     [002] 33979.804841: hrtimer_interrupt <-smp_apic_timer_interrupt

To see which functions are being traced, you can cat the file:

 # cat set_ftrace_filter
hrtimer_interrupt
sys_nanosleep


Perhaps this is not enough. The filters also allow simple wild
cards. Only the following are currently available

  <match>*  - will match functions that begin with <match>
  *<match>  - will match functions that end with <match>
  *<match>* - will match functions that have <match> in it

These are the only wild cards which are supported.

  <match>*<match> will not work.

Note: It is better to use quotes to enclose the wild cards,
      otherwise the shell may expand the parameters into names
      of files in the local directory.

 # echo 'hrtimer_*' > set_ftrace_filter

Produces:
```

```
  <idle>-0     [002] 68988.813277: hrtimer_hres_active <-hrtimer_run_pending
  <idle>-0     [002] 68988.813286: hrtimer_get_next_event <-
get_next_timer_interrupt
  <idle>-0     [002] 68988.813286: hrtimer_hres_active <-hrtimer_get_next_event
  <idle>-0     [002] 68988.813287: hrtimer_start <-tick_nohz_stop_sched_tick
  <idle>-0     [002] 68988.813288: hrtimer_hres_active <-__remove_hrtimer
  <idle>-0     [002] 68988.813288: hrtimer_force_reprogram <-__remove_hrtimer
  <idle>-0     [003] 68988.881182: hrtimer_interrupt <-smp_apic_timer_interrupt
  <idle>-0     [003] 68988.881185: hrtimer_run_queues <-run_local_timers
  <idle>-0     [003] 68988.881186: hrtimer_hres_active <-hrtimer_run_queues
  <idle>-0     [003] 68988.881189: hrtimer_forward <-tick_sched_timer
  <idle>-0     [003] 68988.881190: hrtimer_run_pending <-run_timer_softirq
  <idle>-0     [003] 68988.881191: hrtimer_hres_active <-hrtimer_run_pending


Notice that we lost the sys_nanosleep.

 # cat set_ftrace_filter
hrtimer_restart
hrtimer_start_expires
hrtimer_hres_active
hrtimer_init_sleeper
hrtimer_forward
hrtimer_force_reprogram
hrtimer_get_res
hrtimer_wakeup
hrtimer_init
hrtimer_get_remaining
hrtimer_try_to_cancel
hrtimer_cancel
hrtimer_start
hrtimer_start_range_ns
hrtimer_start_expires
hrtimer_get_next_event
hrtimer_interrupt
hrtimer_peek_ahead_timers
hrtimer_run_pending
hrtimer_run_queues
hrtimer_nanosleep
hrtimer_nanosleep_restart
hrtimer_start_expires.clone.5
hrtimer_forward_now
hrtimer_restart


This is because the '>' and '>>' act just like they do in bash.
To rewrite the filters, use '>'
To append to the filters, use '>>'

To clear out a filter so that all functions will be recorded
again:

 # echo > set_ftrace_filter
 # cat set_ftrace_filter
 #

Again, now we want to append.

 # echo sys_nanosleep > set_ftrace_filter
```

```
 # cat set_ftrace_filter
sys_nanosleep
 # echo 'hrtimer_*' >> set_ftrace_filter
 # cat set_ftrace_filter
hrtimer_restart
hrtimer_start_expires
hrtimer_hres_active
hrtimer_init_sleeper
hrtimer_forward
hrtimer_force_reprogram
hrtimer_get_res
hrtimer_wakeup
hrtimer_init
hrtimer_get_remaining
hrtimer_try_to_cancel
hrtimer_cancel
hrtimer_start
hrtimer_start_range_ns
hrtimer_start_expires
hrtimer_get_next_event
hrtimer_interrupt
hrtimer_peek_ahead_timers
hrtimer_run_pending
hrtimer_run_queues
hrtimer_nanosleep
sys_nanosleep
hrtimer_nanosleep_restart
hrtimer_start_expires.clone.5
hrtimer_forward_now
hrtimer_restart


The set_ftrace_notrace prevents those functions from being
traced.

 # echo '*preempt*' '*lock*' > set_ftrace_notrace

Produces:

# tracer: function
#
#           TASK-PID    CPU#    TIMESTAMP  FUNCTION
#              | |       |          |         |
          <idle>-0     [002] 69247.262737: need_resched <-mwait_idle
          <idle>-0     [002] 69247.262738: test_ti_thread_flag <-need_resched
          <idle>-0     [002] 69247.262739: __exit_idle <-cpu_idle
          <idle>-0     [002] 69247.262740: test_ti_thread_flag <-cpu_idle
          <idle>-0     [002] 69247.262741: enter_idle <-cpu_idle
          <idle>-0     [002] 69247.262742: mwait_idle <-cpu_idle
##### CPU 0 buffer started ####
          <idle>-0     [000] 69247.262742: need_resched <-mwait_idle
          <idle>-0     [002] 69247.262742: need_resched <-mwait_idle
          <idle>-0     [000] 69247.262743: test_ti_thread_flag <-need_resched
          <idle>-0     [002] 69247.262743: test_ti_thread_flag <-need_resched
          <idle>-0     [000] 69247.262744: __exit_idle <-cpu_idle
          <idle>-0     [002] 69247.262744: trace_power_start.clone.5 <-mwait_idle
          <idle>-0     [000] 69247.262745: test_ti_thread_flag <-cpu_idle
          <idle>-0     [002] 69247.262745: need_resched <-mwait_idle
          <idle>-0     [000] 69247.262746: enter_idle <-cpu_idle
```

```
We can see that there's no more lock or preempt tracing.


Dynamic ftrace with the function graph tracer
---------------------------------------------

Although what has been explained above concerns both the
function tracer and the function-graph-tracer, there are some
special features only available in the function-graph tracer.

If you want to trace only one function and all of its children,
you just have to echo its name into set_graph_function:

 echo __do_fault > set_graph_function

will produce the following "expanded" trace of the __do_fault()
function:

 0)               |  __do_fault() {
 0)               |    filemap_fault() {
 0)               |      find_lock_page() {
 0)   0.804 us    |        find_get_page();
 0)               |        __might_sleep() {
 0)   1.329 us    |        }
 0)   3.904 us    |      }
 0)   4.979 us    |    }
 0)   0.653 us    |    _spin_lock();
 0)   0.578 us    |    page_add_file_rmap();
 0)   0.525 us    |    native_set_pte_at();
 0)   0.585 us    |    _spin_unlock();
 0)               |    unlock_page() {
 0)   0.541 us    |      page_waitqueue();
 0)   0.639 us    |      __wake_up_bit();
 0)   2.786 us    |    }
 0) + 14.237 us   |  }
 0)               |  __do_fault() {
 0)               |    filemap_fault() {
 0)               |      find_lock_page() {
 0)   0.698 us    |        find_get_page();
 0)               |        __might_sleep() {
 0)   1.412 us    |        }
 0)   3.950 us    |      }
 0)   5.098 us    |    }
 0)   0.631 us    |    _spin_lock();
 0)   0.571 us    |    page_add_file_rmap();
 0)   0.526 us    |    native_set_pte_at();
 0)   0.586 us    |    _spin_unlock();
 0)               |    unlock_page() {
 0)   0.533 us    |      page_waitqueue();
 0)   0.638 us    |      __wake_up_bit();
 0)   2.793 us    |    }
 0) + 14.012 us   |  }

You can also expand several functions at once:

 echo sys_open > set_graph_function
 echo sys_close >> set_graph_function

Now if you want to go back to trace all functions you can clear
this special filter via:
```

```
 echo > set_graph_function


Outputting the trace on panic or oops
-------------------------------------

The tracer may be used to dump the trace for the oops'ing cpu on
a kernel oops into the system log. To enable this,
ftrace_dump_on_oops must be set. To set ftrace_dump_on_oops, one
can either add "ftrace_dump_on_oops" on the kernel command line
or use the sysctl function or set it via the proc system
interface.

  sysctl kernel.ftrace_dump_on_oops=1

or

  echo 1 > /proc/sys/kernel/ftrace_dump_on_oops


Here's an example of such a dump after a null pointer dereference.

BUG: unable to handle kernel NULL pointer dereference at (null)
IP: [<ffffffff8125022f>] sysrq_handle_crash+0x16/0x20
PGD 3ed76067 PUD 373c5067 PMD 0
Oops: 0002 [#1] PREEMPT SMP
last sysfs file: /sys/devices/system/cpu/cpu3/cache/index1/shared_cpu_map
Dumping ftrace buffer:
---------------------------------
    bash-1570    2..... 22115712us : test_ti_thread_flag <-mnt_want_write
    bash-1570    2..... 22115713us : file_move <-__dentry_open
    bash-1570    2..... 22115714us : _raw_spin_lock <-file_move
    bash-1570    2...1. 22115715us : do_raw_spin_lock <-_raw_spin_lock
    bash-1570    2...1. 22115716us : _raw_spin_unlock <-file_move
    bash-1570    2..... 22115717us : test_ti_thread_flag.clone.2 <-_raw_spin_unlock
    bash-1570    2..... 22115718us : security_dentry_open <-__dentry_open
[...]
  <idle>-0       0d..1. 22118642us : need_resched <-mwait_idle
    bash-1570    2d..1. 22118642us : test_ti_thread_flag <-pagefault_enable
  <idle>-0       3d..1. 22118642us : need_resched <-mwait_idle
  <idle>-0       1...1. 22118642us : __exit_idle <-cpu_idle
  <idle>-0       0d..1. 22118642us : test_ti_thread_flag <-need_resched
    bash-1570    2d..1. 22118643us : oops_enter <-oops_begin
  <idle>-0       3d..1. 22118643us : test_ti_thread_flag <-need_resched
  <idle>-0       0d..1. 22118643us : trace_power_start.clone.5 <-mwait_idle
  <idle>-0       1...1. 22118643us : test_ti_thread_flag <-cpu_idle
---------------------------------
CPU 2
Pid: 1570, comm: bash Not tainted 3.2.16-test #2 0C9316/Precision WorkStation 470
RIP: 0010:[<ffffffff8125022f>]  [<ffffffff8125022f>] sysrq_handle_crash+0x16/0x20
RSP: 0018:ffff880037027e38  EFLAGS: 00010096
RAX: 0000000000000010 RBX: 0000000000000063 RCX: 00000000ffffffe5
RDX: 0000000000000000 RSI: 0000000000000000 RDI: 0000000000000063
RBP: ffff880037027e38 R08: 0000000000000001 R09: ffffffffffffffff
R10: ffff880037027c48 R11: ffffffff819438bc R12: 0000000000000000
R13: ffffffff816ec1c0 R14: 0000000000000003 R15: 0000000000000296
FS:  00007f19e62f7700(0000) GS:ffff880001a80000(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: 0000000000000000 CR3: 000000003eda8000 CR4: 00000000000006e0
```

```
DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400
Process bash (pid: 1570, threadinfo ffff880037026000, task ffff88003c036700)
Stack:
 ffff880037027e88 ffffffff81250662 ffff880037027ea8 ffffffff00000000
<0> fffffffffffffffb 0000000000000002 ffffffff81250701 ffff88003cfc8440
<0> 00007f19e6308000 0000000000000002 ffff880037027ea8 ffffffff81250738
Call Trace:
 [<ffffffff81250662>] __handle_sysrq+0xa3/0x142
 [<ffffffff81250701>] ? write_sysrq_trigger+0x0/0x3e
 [<ffffffff81250738>] write_sysrq_trigger+0x37/0x3e
 [<ffffffff8113d853>] proc_reg_write+0x90/0xaf
 [<ffffffff810f4685>] vfs_write+0xac/0x100
 [<ffffffff810f5591>] ? fget_light+0x40/0x8a
 [<ffffffff810f488e>] sys_write+0x4a/0x6e
 [<ffffffff81002cdb>] system_call_fastpath+0x16/0x1b


trace_pipe
----------

The trace_pipe outputs the same content as the trace file, but
the effect on the tracing is different. Every read from
trace_pipe is consumed. This means that subsequent reads will be
different. The trace is live.

 # echo function > current_tracer
 # cat trace_pipe > /tmp/trace.out &
[1] 4153
 # echo 1 > tracing_on
 # usleep 1
 # echo 0 > tracing_on
 # cat trace
# tracer: function
#
#           TASK-PID   CPU#    TIMESTAMP  FUNCTION
#              | |      |          |         |

 #
 # cat /tmp/trace.out
       <idle>-0     [003]   392.260222: trace_power_start.clone.5 <-mwait_idle
       <idle>-0     [003]   392.260223: need_resched <-mwait_idle
       <idle>-0     [003]   392.260224: test_ti_thread_flag <-need_resched
       <idle>-0     [003]   392.260225: __exit_idle <-cpu_idle
       <idle>-0     [003]   392.260226: test_ti_thread_flag <-cpu_idle
       <idle>-0     [003]   392.260227: enter_idle <-cpu_idle
       <idle>-0     [003]   392.260228: mwait_idle <-cpu_idle
       <idle>-0     [003]   392.260229: need_resched <-mwait_idle
       <idle>-0     [003]   392.260229: test_ti_thread_flag <-need_resched
       <idle>-0     [003]   392.260230: trace_power_start.clone.5 <-mwait_idle
       <idle>-0     [003]   392.260231: need_resched <-mwait_idle
       <idle>-0     [003]   392.260232: test_ti_thread_flag <-need_resched
       <idle>-0     [003]   392.260233: __exit_idle <-cpu_idle


Note, reading the trace_pipe file will block until more input is
added. By changing the tracer, trace_pipe will issue an EOF. We
needed to set the function tracer _before_ we "cat" the
trace_pipe file.
```

```
trace entries
-------------

Having too much or not enough data can be troublesome in
diagnosing an issue in the kernel. The file buffer_size_kb is
used to modify the size of the internal trace buffers. The
number listed is the number of kilobytes each CPU ring buffer has.
To know the full size, multiply the number of possible CPUS with
the size in buffer_size_kb.

 # cat buffer_size_kb
1408

Note, when modifying the ring buffer size, tracing will stop
while the buffer size is being updated, and then will continue
after the update.

 # echo 10000 > buffer_size_kb
 # cat buffer_size_kb
10000

-----------

More details can be found in the source code, in the kernel/trace/*.c files.
```

Versionsgeschichte

Version 3-1.400 **2013-10-31** **Rüdiger Landmann**

Rebuild with publican 4.0.0

Version 3-0 **Mon Jun 11 2012** **Cheryn Tan**

Vorbereitet für Veröffentlichung (MRG 2.2).

Version 2-8 **Fri Jun 1 2012** **Cheryn Tan**

BZ#813890 - Dokumentation über /dev/cpu_dma_latency hinzugefügt.

Version 2-6 **Wed May 16 2012** **Cheryn Tan**

BZ#809309 - Weitere Änderungen an kdump-Anweisungen vorgenommen.
BZ#821697 - Obsoleten Hinweis auf bdflush entfernt.

Version 2-5 **Tue May 15 2012** **Cheryn Tan**

BZ#809309 - kdump-Anweisungen entsprechend technischer Prüfung überarbeitet.
BZ#813890 - Abschnitt über Verwendung von _COARSE-Uhren im Kapitel über Applikationsoptimierung hinzugefügt.
Abschnitt "MRG Realtime spezifische gettimeofday-Optimierung" entfernt.

Version 2-4 **Thu May 10 2012** **Cheryn Tan**

BZ#804847 - Link zu RHEL-Netzwerkdokumentation hinzugefügt.
BZ#805746 - Link zu Infiniband-Anweisungen hinzugefügt.
BZ#813890 - gettimeofday-Optimierung entfernt, Abschnitt über Hardware-Uhren und Timestamps hinzugfügt.
BZ#800737 - ftrace-Anhang mit Kernel-Änderungen aktualisiert.

Version 2-3 **Thu May 3 2012** **Cheryn Tan**

BZ#804853 - Kurze Übersicht über HPN hinzugefügt.
BZ#804847 - Kurze Übersicht über RoCEE hinzugefügt.
BZ#809309 - kdump-Anweisungen für RHEL6 aktualisiert.
BZ#800737 - Referenzen auf MRG RT Kernel auf 3.2 aktualisiert.

Version 2-1 **Tue Feb 28 2012** **Tim Hildred**

Konfigurationsdatei für neues Publikationstool.

Version 2-0 **Wed Dec 7 2011** **Alison Young**

Vorbereitet für Veröffentlichung

Version 1-7 **Wed Nov 16 2011** **Alison Young**

BZ#752406 - RHEL-Versionen geändert

Version 1-5 **Tue Oct 12 2011** **Alison Young**

BZ#716559 - Ereignis- und Funktions-Tracer aktualisiert

Version 1-3 **Tue Oct 11 2011** **Alison Young**

BZ#717261 - Inkorrekte Daten
BZ#725667 - trace-cmd in RHEL 6

Version 1-2	Wed Oct 5 2011	Alison Young

BZ#712267 - Link zu nicht vorhandener Mailingliste

Version 1-1	Thu Sep 22 2011	Alison Young

Versionsnummerierung geändert

Version 1-0	Thu Jun 23 2011	Alison Young

Vorbereitet für Veröffentlichung

Version 0.1-5	Thu June 02 2011	Alison Young

Neuerstellung nach Aktualisierung von Brand-Paket

Version 0.1-4	Mon May 23 2011	Alison Young

Änderungen nach technischer Überprüfung

Version 0.1-3	Mon May 16 2011	Alison Young

BZ#584297 - Abschnitte über Latenz-Tracing neu aufgebaut
BZ#666962 - Aktualisierung für RHEL6

Version 0.1-2	Thu Apr 05 2011	Alison Young

Kleinere Aktualisierungen

Version 0.1-1	Tue Apr 05 2011	Alison Young

BZ#683586 - Abschnitt über weitere Informationsquellen aktualisiert
Kleinere XML-Aktualisierungen

Version 0.1-0	Wed Feb 23 2011	Alison Young

Abgezweigt von 1.3